0x74	SrclkPre	14	0x0000	Number of *pclk* cycles between *phi_lsyncl* falling edge and *phi_srclk* pulse generation, or printhead data transfer
0x78	SrclkPost	14	0x0000	Number of *pclk* cycles allowed margin from last *srclk* pulse in a line to before next line sync
0x7C-0x80	PrintHeadRate[1:0]	2x16	0xFFFF	Specifies the active to inactive ratio of *phi_srclk* for the printhead ICs. A 1 indicates Active. Bus 0 - Printhead IC channel A Bus 1 - Printhead IC channel B
0x84	DotOrderMode	1	0x0	Specifies the dot transmit order to the printhead Channel A. Printhead Channel B is always the opposing order. 0 - Even before Odd dots 1 - Odd before Even dots
Fire Control				
0x98	FrclkPre	14	0x0000	Number of *pclk* cycles after lsyncl transitions from 0 to 1 to *phi_frclk* pulse generation
0x9C	FrclkLow	14	0x0000	Number of *pclk* cycles *phi_frclk* should remain low.
0xA0	FrclkHigh	14	0x0000	Number of *pclk* cycles *phi_frclk* should remain high.
0xA4	FrclkNum	16	0x0000	Number of *phi_frclk* pulses per line time.
0xA8	FireGenSoftTrigger	1	0x0	Only active when *PrintHeadCpuCtrlMode* is set to 1, *PrintHeadCpuCtrl* is 1 and pin is in output mode. Bit 0 controls *frclk* generator. A 0 to 1 transition on a bit triggers the corresponding generator to create the programmed pulse profile (configured by *FrclkNum,FrclkHigh,FrclkLow,FrclkPre* registers) when complete the bit gets

				reset to 0.
Working Registers				
0xAC-0xB0	LineDotCnt	2x16	0x0000	Indicates the number of dot processed in the current line Bus 0 - Printhead Channel A Bus 1 - Printhead Channel B (Read Only Registers)

The configuration registers in the PHI block are clocked at *pclk* rates but some blocks in the PHI are clocked by different and asynchronous clocks. Configuration values are not re-synchronized, it is therefore important that the *Go* register be set to zero while updating configuration values. This prevents logic from entering unknown states due to metastable clock domain transfers.

5   Some registers can be written to at any time such as the direct CPU control registers (*PrintHeadCpuIn*, *PrintHeadCpuDir*, *PrintHeadCpuOut* and *PrintHeadCpuCtrl*), the *Go* register and the *PrintStart* register. All registers can be read from at any time.

32.9.4   Dot counter

The dot counter keeps a running count of the number of dots fired for each color plane. The

10   counters are 32 bits wide and will saturate. When the CPU wants to read the dot count for a particular color plane it must write to the *DotCountSnap* register. This causes all 6 running counter values to be transferred to the *DotCount* registers in the configuration registers block. The running counter values are reset.

```
                // reset if being snapped
15              if (dot_cnt_snap == 1) then{
                    dot_count[5:0]        = accum_dot_count[5:0]
                    accum_dot_count[5:0]  = 0
                    }
                // update the counts
20              for (color=0;color < 6;color++) {
                    if (accum_dot_count[color] != 0xffff_ffff) {
                        // data valid, first dot stream
                        data_valid    =    ((phi_llu_ready[0]    ==    1)    AND
                (llu_phi_avail[0] == 1))
25                      if  ((data_valid == 1)  AND  (llu_phi_data[0][color]  ==
                1)) then
                            accum_dot_count[color] ++
                        // data valid, second dot stream
                        data_valid    =    ((phi_llu_ready[1]    ==    1)    AND
30              (llu_phi_avail[1] == 1))
                        if  ((data_valid == 1)  AND  (llu_phi_data[1][color]  ==
                1)) then
                            accum_dot_count[color] ++
                    }
35              }
```

32.9.5   Sync generator

The sync generator logic has two modes of operation, master and slave mode. In master mode (configured by the *PhiMode* register) it generates the *lsyncl_o* output based on configured values and control triggers from the PHI controller. In slave mode it de-glitches the incoming *lsyncl_i* signal, and filters the *lsyncl* signal with the minimum configured period.

5 After reset or a pulse on *phi_go_pulse* the machine returns to the *Reset* state, regardless of what state it's currently in.

The state machine waits until it's enabled (*sync_en==1*) by the PHI controller state machine. When enabled it can proceed to the *SyncPre* or *SyncWait* depending on whether the state machine is configured in master or slave mode. In master mode it generates the *lsyncl* pulses, in

10 slave mode it receives and filters the *lsyncl* pulses from the master sync generator.

On transition to the *SyncPre* state a counter is loaded with the *LsyncPre* value, and while in the *SyncPre* the counter is decremented. When the count is zero the machine proceeds to the *SyncLow* state loading the counter with *LsyncLow* value.

The machine waits in the *SyncLow* state until the counter has decremented to zero. It proceeds to

15 the *SyncHigh* state pulsing the *line_st* signal on transition and counts *LsyncHigh* number of cycles. This indicates to the PHI controller the line start aligned to the *lsyncl* positive edge. While in *LsyncLow* state the *lsyncl_o* output is set to 0 and in *SyncHigh* the *lsyncl_o* output is set to 1. When the count is zero and the current line is not the last (*last_line == 0*), the machine returns to the *SyncLow* state to begin generating a new line sync pulse. The transition pulses the *line_fin*

20 signal to the PHI controller.

The loop is repeated until the current line is the last (*last_line ==1*), and the machine returns to the *Reset* state to wait for the next page start.

In slave mode the state machine proceeds to the *SyncWait* state when enabled. It waits in this state until a *lsync_pulse_rise* is received from the input de-glitch circuit. When a pulse is detected

25 the machine jumps to the *SyncPeriod* state and begins counting down the *LsyncHigh* number of clock cycles before returning to the *SyncWait* state. Note in slave mode the *LsyncHigh* specifies the minimum number of *pclk* cycles between *Lsync* pulses. On transition from the *SyncWait* to the *SyncPeriod* state the *line_st* signal to the PHI controller is pulsed to indicate the line start. While in the *SyncPeriod* state if a *lsync_pulse_fall* is detected the state machine will signal a sync error

30 (via *sync_err*) to the PHI controller and cause a buffer underrun interrupt.

32.9.5.1 Lsyncl input de-glitch

The *lsync_i* input is considered an asynchronous input to the PHI, and is passed through a synchronizer to reduce the possibility of metastable states occurring before being passed to the de-glitch logic.

35 The input de-glitch logic rejects input states of duration less than the configured number of clock cycles (*lsync_deglitch_cnt*), input states of greater duration are reflected on the output, and are negative and positive edge detected to produce the *lsync_pulse_fall* and *lsync_pulse_rise* signal to the main generator state machine. The counter logic is given by

```
if ( lsync_i != lsync_i_delay) then
    cnt       = lsync_deglitch_cnt
```

40

```
                        output_en = 0
                elsif (cnt == 0 ) then
                    cnt         = cnt
                    output_en = 1
                else
                    cnt --
                    output_en = 0
```

32.9.5.2 Line Sync Interrupt logic

The line sync interrupt logic counts the number of line syncs that occur (either internally or externally generated line syncs) and determines whether to generate an interrupt or not. The number of line syncs it counts before an interrupt is generated is configured by the *LineSyncInterrupt* register. The interrupt is disabled if *LineSyncInterrupt* is set to zero.

```
                // implement the interrupt counter
                if (phi_go_pulse ==1) then
                    line_count = 0
                elsif (line_st == 1) AND (line_count == 0)) then
                        line_count = linecount_int
                elsif ((line_st == 1) AND (line_count != 0)) then
                        line_count --
                // determine when to pulse the interrupt
                if (linesync_int == 0 ) then // interrupt disabled
                    phi_icu_linesync_int = 0;
                elsif ((line_st == 1) AND (line_count == 1)) then
                    phi_icu_linesync_int = 1
```

32.9.6   Fire generator

The fire generator block creates the signal profile for the *phi_frclk* signal to the printhead. The *frclk* is based on configured values and is timed in relation to the *fire_st* pulse from the PHI controller block. Should the *phi_frclk* state machine receive a *fire_st* pulse before it has completed the sequence the machine will restart regardless of its current state.

Alternatively the frclk state machine can be triggered to generate their configured pulse profile by software. A low to high transition on the *FireGenSoftTrigger* register will cause a pulse on *soft_frclk_st* triggering the state machine to begin generating the pulse profile. When the state machine has completed its sequence it will clear the *FireGenSoftTrigger* register bit (via *soft_fire_clr* signal). The *FireGenSoftTrigger* register will only be active when the printhead interface is in CPU direct control mode (*PrintHeadCpuCtrl* = 1) , the fire generator is in software trigger mode (*PrintHeadCpuCtrlMode[x]* = 1) and the pin is configured to be output mode (*PrintHeadCpuDir[x]* = 1).

The fire generator consists of a state machine for creating the *phi_frclk* signal. The *phi_frclk* signal is generated relative to the *lsyncl* signal.

The machine is reset to the *Reset* state when *phi_go_pulse* ==1 or the reset is active, regardless of the current state.

The machine waits in the reset state until it receives a *fire_st* pulse from the PHI controller (or an *soft_fire_st* from the configuration registers). The controller will generate a *fire_st* pulse at the

534

beginning of each dot line. On the state transition the cycle counter is loaded with the *FrclkPre* value and the repeat counter is loaded with the *FrclkNum* value.

The state machine waits in the *FirePre* state until the cycle counter is zero, after which it jumps to the *FireHigh* state and loads the cycle counter with *FrclkHigh* value. Again the state machine waits until the count is zero and then proceeds to the *FireLow* state. On transition the cycle counter is loaded with the *FireLow* value. The state machine waits in the *FireLow* state while the cycle counter is decremented.

When the cycle counter reaches zero and the *repeat_count* is non-zero, the *repeat_count* is decremented, the cycle counter is loaded with the *FrclkHigh* value and the state machine jumps to the *FireHigh* state to repeat the *phi_frclk* generation cycle. The loop is repeated until the *repeat_count* is zero. In such cases the state machine goes to the reset state resetting *FireGenSoftTrigger* (via the *soft_fire_clr* signal) register on the transition and waits for the next *fire_st* pulse.

When in the *Reset* state the *fire_rdy* signal is active to indicate to the controller that the fire generator is ready.

32.9.7   PHI controller

The PHI controller is responsible for controlling all functions of the PHI block on a line by line basis. It controls and synchronizes the sync generator, the fire generator, and datapath unit, as well as signalling back to the CPU the PHI status. It also contains a line counter to determine when a full page has completed printing.

The PHI controller state machine is reset to *Reset* state by a reset or *phi_go_pulse* == 1.

It will remain in reset until the block is enabled by *phi_go* == 1. Once enabled the state machine will jump to the *FirstLine* state, trigger the transfer of one line of data to the printhead (*data_st* == 1) and the line counter will be initialized to the page length (*PageLenLine*). Once the line is transferred (*data_fin* from the datapath unit) the machine will go to *Printstart* state and signal the CPU using an interrupt that the PHI is ready to begin printing (*phi_icu_print_rdy*). The line counter will also be decremented. It will then wait in the *Printstart* state until the CPU acknowledges the print ready signal and enables printing by writing to the *PrintStart* register.

The state machine proceeds to the *SyncWait* state and waits for a line start condition (*line_st* ==1). The line start condition is different depending on whether the PHI is configured as being in a master or slave SoPEC (the *PhiMode* register). In either case the sync generator determines the correct line start source and signals the PHI controller via the *line_st* signal. Once received the machine proceeds to the *LineTrans* state, with the transition triggering the fire generator to start (*fire_st*), the datapath unit to start (*data_st*) and the sync generator to start (*sync_st*).

While in the *LineTrans* state the fire, sync and datapath unit will be producing line data. When finished processing a line the datapath unit will assert the line finished (*data_fin*) signal. If the line counter is not equal to 1 (i.e. not the last line) the state machine will jump back to the *SyncWait* state and wait for the start condition for the next line. The line counter will be decremented. If the line counter is one then the machine will proceed to the *LastLine* state.

The *LastLine* state generates one more line of fire pulses to print the last line held in the shift registers of the printhead. Once complete (*fire_fin* ==1) the state machine returns to the reset state and waits for the next page of data. On page completion the state machine generates a *phi_icu_page_finish* interrupt to signal to the CPU that the page has completed, the

5   *phi_icu_page_finish* will also cause the *Go* register to reset automatically.

While the state machine is in the *LineTrans* state (or in *FirstLine* state and the PHI is in slave mode) and waiting for the datapath unit to complete line processing, it is possible (e.g. an excessive PEP stall) that a line finish condition occurs (*line_fin* == 1) but the datapath unit is not ready. In this case an underrun error is generated. The state machine goes to the *Underrun* state

10   and generates a *phi_icu_underrun* interrupt to the CPU. The PHI cannot recover from a buffer underrun error, the CPU must reset the PEP blocks and re-start printing. The *phi_icu_underrun* will also cause the *Go* register to reset automatically.

32.9.8   CPU IO control

The CPU IO control block is responsible for providing direct CPU control of the IO pins via the

15   configuration registers. It also accepts the input signals from the printhead and re-synchronizes them to the *pclk* domain, and debug signals from the RDU and muxes them to output pins.

Table   contains the direct mapping of configuration registers to printhead IO pins. Direct CPU control is enabled only when *PrintHeadCpuCtrl* is set to one. In normal operation (i.e. *PrintHeadCpuCtrl* == 0) the printhead *frclk* pin is always in output mode (*phi_frclk_e*=1), the

20   *phi_lsyncl* will be in output if the SoPEC is the master, i.e. *phi_lsyncl_e = phi_mode*, and *readl* will be set high.

The *PrintHeadCpuCtrlMode* register determine whether the *frclk* pin should be driven by the fire generator logic or direct from the CPU *PrintHeadCpuOut* register.

The pseudocode for the CPU IO control is:

```
25            if (printhead_cpu_ctrl == 1) then   // CPU access enabled
                 // outputs
                 if  (PrintHeadCpuCtrlMode[0]   ==  1)  then         //  fire
              generator controlled
             ·    phi_frclk_o        = frclk
30               else                                              // normal
              direct CPU control
                    phi_frclk_o        = printhead_cpu_out[1]
                 phi_ph_data_o[0][1:0] = printhead_cpu_out[4:3]
                 phi_ph_data_o[1][1:0] = printhead_cpu_out[6:5]
35               phi_srclk[1:0]        = printhead_cpu_out[8:7]
                 phi_readl             = printhead_cpu_out[9]
                 // direction control
                 phi_lsyncl_e          = printhead_cpu_dir[0]
                 phi_frclk_e           = printhead_cpu_dir[1]
40               // input assignments
                 printhead_cpu_in[0]   = synchronize(phi_lsyncl_i)
                 printhead_cpu_in[1]   = synchronize(phi_frclk_i)
              else   // normal connections
```

```
                // outputs
                phi_ph_data_o[0][1:0]   = ph_data[0][1:0]
                phi_ph_data_o[1][1:0]   = ph_data[1][1:0]
                phi_lsyncl_o            = lsync_o
                phi_readl               = 1
                phi_srclk[1:0]          = srclk[1:0]
                phi_frclk_o             = frclk
                // direction control
                phi_frclk_e             = 1
                phi_lsyncl_e                = phi_mode      // depends on Master
        or Slave mode
                // inputs
                lsyncl_i                            = phi_lsync_i  // connected
        regardless
                // debug overrides any other connections
                if (debug_cntrl[0] == 1) then
                    phi_frclk_o         = debug_data_valid
                    phi_frclk_e         = 1
                    phi_readl           = pclk
```

The debug signalling is controlled by the RDU block (see Section 11.8 Realtime Debug Unit (RDU)), the IO control in the PHI muxes debug data onto the PHI pins based on the control signals from the RDU.

32.9.9   Datapath Unit

32.9.10   Dot order controller

The dot order controller is responsible for controlling the dot order blocks. It monitors the status of each block and determines the switch over point, at which the connections from odd and even dot streams to printhead channels are swapped.

The machine is reset to the *Reset* state when *phi_go_pulse* == 1 or the reset is active. The machine will wait until it receives a *data_st* pulse from the PHI controller before proceeding to the *LineStart* state. On the transition to the *LineStart* state it will reset the dot counter in each dot order block via the *dot_cnt_rst* signal.

While in the *LineStart* state both dot order blocks are enabled (*gen_en*==1). The dot order blocks process data until each of them reach their mid point. The mid point of a line is defined by the configured printhead size (i.e. *print_head_size*). When a dot order block reaches the mid point it immediately stops processing and waits for the remaining dot order block. When both dot order blocks are at the mid point (*mid_pt* == 11) the controller clocks through the *LineMid* state to allow the pipeline to empty and immediately goes to *LineEnd* state.

In the *LineEnd* state the *mode_sel* is switched and the dot order blocks re-enabled, in this state the dot order blocks are reading data from the opposite LLU dot data stream as in *LineStart* state.

The controller remains in the *LineEnd* state until both dot order blocks have processed a line i.e. *line_fin* == 11.

On completion of both blocks the controller returns to the *Reset* state and again awaits the next *data_st* pulse from the PHI controller. When in *Reset* state the machine signals the PHI controller that it's ready to begin processing dot data via the *dot_order_rdy* signal.

The dot order controller selects which dot streams should feed which printhead channels. The order can be changed by configuring the *DotOrderMode* register. In all cases Channel A and Channel B must be in opposing dot order modes. Table 216 shows the possible modes of operation.

Table 216. Mode selection in Dot order controller.

Channel	Mode_sel	DotOrderMode	Dot transmit order
A	0	0	Even before Odd (EBO mode), even dot stream feeds Channel A printhead, first half line.
	0	1	Odd before Even (OBE mode), odd dot stream feeds Channel A printhead, first half line.
	1	0	Even before Odd (EBO mode), even dot stream feeds Channel A printhead, second half line.
	1	1	Odd before Even (OBE mode), odd dot stream feeds Channel A printhead, second half line.
B	0	0	Odd before Even (OBE mode), odd dot stream feeds Channel B printhead, second half line
	0	1	Even before Odd (EBO mode), even dot stream feeds Channel B printhead, second half line.
	1	0	Odd before Even (OBE mode), odd dot stream feeds Channel B printhead, first half line.
	1	1	Even before Odd (EBO mode), even dot stream feeds Channel B printhead, first half line.

32.9.10.1 Dot order unit

The dot order control accepts dot data from either dot stream from the LLU and writes the dot data into the dot buffer. It has two modes of operation, odd before even (OBE) and even before odd (EBO). In the OBE mode data from the odd stream dot data is accepted first then even, in EBO mode it's vice versa. The mode is configurable by the *DotOrderMode* register.

The dot order unit maintains a dot count that is decremented each time a new dot is received from the LLU. The dot order controller resets the dot counter to the *print_head_size[15:0]* at the start of a new line via the *dot_cnt_rst* signal. The dot count is compared with the printhead size (*print_head_size[15:0]* divided by 2) to determine the mid point (*mid_pt*) and the line finish point

5   (*line_fin*) when the dot counter is zero.

The mid point is defined as the half the number of dots in a particular printhead, and is derived from the the *print_head_size* bus by dividing by 2 and rounding down.

```
                // define the mid point
                if (dot_cnt[15:0] == print_head_size[15:1] )then
10                 mid_pt = 1
                else
                   mid_pt = 0
```

The dot order unit logic maintains the dot data write pointer. Each time a new dot is written to the dot buffer the write pointer is incremented. The fill level of the dot buffer is determined by

15   comparing the read and write pointers. The fill level is used to determine when to backpressure the LLU (*ready* signal) due to the dot buffer filling. A suitable threshold value is determined to allow for the full LLU pipeline to empty into the dot buffer.

The dot order stalling control is given by:

```
                // determine the ready/avail signal to use, based on mode
20              select
                if (mode_sel == 1) then
                   dot_active = llu_phi_avail[0] AND ready
                   wr_data    = llu_phi_data[0]
                else
25                 dot_active = llu_phi_avail[1] AND ready
                   wr_data    = llu_phi_data[1]
                // update the counters
                if (dot_active == 1) then {
                   wr_en = 1
30                 wr_adr ++
                   if (dot_cnt == 0) then
                      dot_cnt = print_head_size
                   else
                      dot_cnt--
35              }
```

The dot writer needs to determine when to stall the LLU dot data stream. A number of factors could stall the dot stream in the LLU such as buffer filling, waiting for the mid point, waiting for the line finish or the dot order controller is waiting for the line start condition from the PHI controller.

The stall logic is given by:

```
40              // determine when to stall the LLU generator
                fill_level = wr_adr - rd_adr
                if (fill_level > (32 - THRESHOLD ))then    // THRESHOLD is
                open value
```

```
        ready = 0                                    // buffer is close
     to full
     elsif ( gen_en == 0) then
        ready = 0                                    // stalled by the
     datapath controller
     else
        ready = 1                                    // everything good
     no stall
```

32.9.10.2 Data generator

The data generator block reads data from the dot buffer and feeds dot data to the printhead at a
configured rate (set by the *PrintheadRate*). It also generates the margin zero data and aligns the
dot data generation to the synchronization pulse from the PHI controller.

The data generator controller waits in *Reset* state until it receives a line start pulse from the PHI
controller (*data_st* signal). Once a start pulse is received it proceeds to the *SrclkPre* state loading
a counter with the *SrclkPre* value. While in this state it decrements the counter. No data is read or
output at this stage. When the count is zero the machine proceeds to the *DataGen1* state.

On transition it loads the counter with the printhead size (*print_head_size*). If margining is to be
used then the configured *print_head_size* should be adjusted by the dot margin value i.e.
print_head_size = (*physical_print_head_size* - (*dot_margin* * 2)).

Dot data is transferred to the printhead serializer in dot-pairs, with one dot-pair transferred every 3
pclk cycles. To construct a dot data pair the state machine reads one dot in the *DataGen1* state,
one dot in the *DataGen2* state and waits for one clock cycle in the *DataGen3* while the data is
transferred to the data serializer. The counter will decrement for every dot data word transferred.
The exact data rate is dictated by the dot buffer fill levels and the configured printhead rate
(*PrintheadRate*). When in *DataGen3* state the machine determines if it should waits for 3 cycles or
transfer another dot pair to the data serializer. The generator determines the rate by comparing
the rate counter (*rate_cnt*) with the configured *PrintheadRate* value. If the bit selected by the
rate_cnt in the *print_head_rate* bus is one data is transferred, otherwise the 3 cycles are skipped
(*Wait1,Wait2* and *Wait3*). If the *PrintHeadRate* is set to all zeros then no data will ever get
transferred. The rate counter is decremented (*rate_cnt*) while in the *DataGen2* and *Wait2* states.
The rate counter is allowed to wrap normally.

The pseudo-code for the rate control *DataGen3* (or *Wait3*) state is given by:
```
     // decrement the rate count
     rate_cnt --                                     // happens in DataGen2, or
     Wait2
     // determine if data should be read
     // first determine if data is available in buffer
     if (rd_adr != wr_adr ) then
        if (print_head_rate[rate_cnt] == 1 ) then
           dot_active = 1
           gate_srclk = 1
           count --
           next_state = DataGen1
```

```
        else
            dot_active = 0
            gate_srclk = 0
            next_state = Wait1
        else
            dot_active = 0
            gate_srclk = 0
            next_state = Wait1
```

When the dot counter reaches zero the state machine will jump to the *MarginGen1* state if the configured margin value is non-zero, otherwise it will jump directly to the *SrclkPost* state. On transition to *MarginGen1* state it loads the cycle counter with the *dot_margin* value, and begins to count down. While in the *MarginGen1,MarginGen2* and *MarginGen3* state machine loop the data generator logic block writes dot data to the printhead but does not read from the dot buffers. It creates zero dot data words for the margin duration. As with normal dot data, it creates one dot in *MarginGen1* and *MarginGen2* states, then wait a clock cycle to allow the transfer to the data serializer to complete.

When the counter reaches zero the machine jumps to the *SrclkPost* state, loads the clock counter with the *SrclkPost* value and decrements. When the count is finished the state machine returns to the *Reset* and awaits the next start pulse. Should a line sync arrive before the data generators have completed (*data_fin* signal) the PHI controller will detect a print error and stall the PHI interface.

As a consequence of the data transfer mechanism of dot pair cycles followed by a wait state, the printhead size (*print_head_size*) and dot margin (*dot_margin*) must always be even dot values.

32.9.10.3 Data serializer

The data serializer block converts 12-bit dot data at *pclk* rates (nominally 160 MHz) to 2-bit data at *doclk* rates (nominally 320 MHz).

The *srclk* is only active when data is available for transfer to the printhead, as enabled by the *gate_srclk* signal. The data rate mechanism in the data generator block will mean that data is not transferred to the printhead on every set of 3 *pclk* cycles. Both the *dot_data* and *gate_srclk* signals are controlled by the data generator block and can only change on a fixed 3 *pclk* cycle boundary. Data is transferred to the printhead on both edges of *srclk* (i.e double data rate DDR). Directly after a line sync pulse the mux control logic and the *srclk* generation logic are reset to a known state (the *srclk* is set high). Before data can begin transfer to the printhead it must generate a line setup edge on *srclk*, causing *srclk* to go low. The line setup edge happens *SrclkPre* number of *pclk* cycles after the line sync falling edge (indicated by the *sr_init* signal from the data generator block).

All data transfers to the printhead will be in groups of 6 2-bit data words, each word clocked on an edge of *srclk*. For each group *srclk* will start low and end low.

At the end of a full line of data transfer the *srclk* must generate a line complete edge to return the *srclk* to a high state before the next line sync pulse. The data generator block generates a *sr_com*

signal to indicate that the data transfer to the printhead has completed and that the line complete edge can be inserted. The *sr_com* signal is generated before the *SrClkPost* period.

The data serializer block allows easy separation of clock gating and clock to logic structures from the rest of the PHI interface.

5 The mux logic determines which data bits from the *dot_data* bus should be selected for output on the *ph_data* bus to the printhead. The mux selector is initialized by an edge detect on the *sr_init* signal from the data generator.

```
                    // determine wrap and init points
                    if (phi_serial_order == 1) then
10                     mux_wrap = 5
                       mux_init = 0
                    else
                       mux_wrap = 0
                       mux_init = 5
15                  // the mux selector logic
                    if ((sr_init_edge == 1)OR( mux_sel == mux_wrap )) then
                       mux_sel = mux_init
                    elsif ( phi_serial_order == 1 ) then
                       mux_sel--                        // decrement order
20                  else
                       mux_sel++                        // increment order
```

The dot data serialization order can be configured by *PhiSerialOrder* register. If the *PhiSerialOrder* is zero the order is *dot[1:0]*, then *dot[3:2]* then *dot[5:4]*. If the register is one then

25 the order is *dot[5:4]*, *dot[3:2]*, *dot[1:0]*.

The srclk control logic is initialized to 1 when a *line_st* positive edge is detected. If either *sr_com_edge*, *sr_init_edge* or *gate_srclk* are equal to one *srclk* is transitioned. *srclk* is always clocked out to the output pins on the negative edge of *doclk* to place the clock edge in the centre of the data.

30 The pseudo code for the control logic is:

```
                    if (line_st_edge ==1 ) then
                       srclk_gen = 1
                    elsif   ((gate_srclk    ==1)    OR    (sr_init_edge==1)    OR
                    (sr_com_edge==1)) then
35                     srclk_gen = ~srclk_gen
                    else
                       // hold
```

33 PACKAGE AND TEST

40 Test Units

33.1   JTAG INTERFACE

A standard JTAG (Joint Test Action Group) Interface is included in SoPEC for Bonding and IO testing purposes. The JTAG port will provide access to all internal BIST (Built In Self Test) structures.

33.2   SCAN TEST I/O

The SoPEC device will require several test IO's for running scan tests. In general scan in and scan out pins will be multiplexed with functional pins.

33.3   ANALOG TEST UNITS

33.3.1   USB PHY Testing

The USB phy analog macro, will contain built-in in test structure, which can be access by either the CPU or through the JTAG port.

33.3.2   Embedded PLL Testing

The embedded clock generator PLL will require test access from JTAG port.

34   SoPEC Pinning and Package

34.1   OVERVIEW

It is intended that the SoPEC package be a 100 pin LQFP. Any spare pins in the package may be used by increasing the number of available GPIO pins or adding extra power and ground pin. The pin list shows the minimum pin requirement for the SoPEC device.

Table 217. SoPEC Pin List (100 LQFP)

Group	Pin Name	#pin s	Dir	Type	Volt	I/O Rate (S/D)	Freq (Mh z)	Description	IO Cell Type	Test Function	Test Macro Function
Clocks and resets											
Group 1	Xtalin	1	I		N/A	N/A	32	Crystal Input pin	AINSA_PM_A	None	
	Xtalout	1	O		N/A	N/A	32	Crystal output pin	ABNST_PM_A	None	
Group 2	reset_n	1	I	LVTT L	3.3v	s	10	Asynchron ous active low reset	IT33LTPUT_PM_A	LT (leakage test)	
PrintHead Interface											
Group 3	phead_dat a	8	O	LVDS	1.5v	d	160	Print head data	OLVDS15_PM_A	None	
	Srclk	4	O	LVDS	1.5v	d	160	Print head clock	OLVDS15_PM_A	None	
Group 4	Readl	1	O	LVTT L	3.3v	s	160	Common Print head mode control	BT3365T_PM_A	A_Clock	

	Frclk	1	I/O	LVTTL	3.3v	s	160	Common Fire pattern shift clock, needs to toggle once per fire cycle	BT3365T_PM_A	B_Clock	
	phi_spare	1	I/O	LVTTL	3.3v	s	160	PHI spare pin (old profile pin)	BT3365T_PM_A	C_Clock1	
	Lsyncl	1	I/O	LVTTL	3.3v	s	160	Line Sync output from Master to Slaves	BT3365T_PM_A	C_Clock2	
USB Connections											
Group 5	Usb_hostd	2	I/O	Differential	3.3v	s	12	USB differential data for host	BUSB2_PM_A	None	
	Usb_devd	2	I/O	Differential	3.3v	s	12	USB differential data for device	BUSB2_PM_A	None	
Group 6	usbd_vbus_sense	1	I	LVTTL	3.3v	s	10	USB device VBUS power sense	BT3365T_PM_C	1 scan out	
	usbd_pull_up_en	1	O	LVTTL	3.3v	s	10	USB device termination enable	BT3365T_PM_C	1 scan out	
JTAG											
Group 7	Tdo	1	O	LVTTL	3.3v	s	10	JTAG Test data out port	BT3365T_PM_A	C_Clock3	
	Tms	1	I	LVTTL	3.3v	s	10	JTAG Test mode select	IT33RIT_PM_A	RI	
	Tdi	1	I	LVTTL	3.3v	s	10	JTAG Test data in port	IT33D1PUT_PM_A	DI1	
	Tck	1	I	LVTTL	3.3v	s	10	JTAG Test	IT33D2PUT_	DI2	

				L				access port clock	PM_A		
General Purpose IO											
Group 8	Gpio[3:0]	4	I/O	LVTTL	3.3v	s	32	ISI interface pins / GPIO	BT3335PUT_PM_B	4 Scanin	
Group 9	Gpio[7:4]	4	I/O	High Drive LVTTL	3.3v	s	32	LED driver pins / general purpose Input/Output	BT3365T_PM_C	4 Scanin	PCNT PROGSROM OSC
Group 10	Gpio[19:8]	12	I/O	LVTTL	3.3v	s	32	General purpose Input/Output	BT3365PUT_PM_B	2 Scanin 10 Scanout	DIAGOUT (aka MRSTR0)
Group 11	Gpio[22:20]	3	I/O	LVTTL	3.3v	s	32	General purpose Input/Output	BT3365PUT_PM_B	CE0_Scan TESTM3 TSTN1	
Group 12	Gpio[31:23]	10	I/O	LVTTL	3.3v	s	32	Functional Spare IOs required for scan test	BT3365T_PM_C	6 Scanin 4 scanout	
Analog Power IO											
Group 13	agnd	1	I	Power	N/A	N/A	N/A	PLL analog gnd	AINSD3_PM_A	None	
	avdd	1	I	Power	N/A	N/A	N/A	PLL analog vdd	AINSD3_PM_A	None	
	agnd	1	I	Power	N/A	N/A	N/A	Oscillator analog gnd	AINSD_PM_A	None	
	avdd	1	I	Power	N/A	N/A	N/A	Oscillator analog vdd	AINSD_PM_A	None	
Test Only Pin											
Group 14	TE	1	I	CMOS	1.5v	N/A	N/A	Test Enable	IC15TEPDT_PM_A	Test only	
	VPP	1	I	CMOS	1.5v	N/A	N/A	Fat Wire Analog Receiver/D	DRAMVPP_PM	Test only	

								river for Embedded DRAM Analog Inputs			
	VWP	1	I	CMOS	1.5v	N/A	N/A	Fat Wire Analog Receiver/Driver for Embedded DRAM Analog Inputs	DRAMVWP_PM	Test only	
	VREFX	1	I	CMOS	1.5v	N/A	N/A	Fat Wire Analog Receiver/Driver for Embedded DRAM Analog Inputs	DRAMVREFX_PM	Test only	
	DLT	1	I	CMOS	1.5v	N/A	N/A	DRAM Iddq Test	IC15DLTPUT_PM	Test only	
	MC	1	I	CMOS	1.5v	N/A	N/A	IO Mode Control	IC15MCT_PM_A	Test only	
	DRAM_EN	1	I	CMOS	1.5v	N/A	N/A	DRAM Enable(EN)	IC15LTPUT_PM_A	Test only	
Total Signal Pins	73		Functional pin count is 62						Test IO count 51		
Power Only Pins											
Group 15	Gnd	8	I	Power	N/A	N/A	N/A	gnd	GND_PM_A	None	
	Vdd	4	I	Power	N/A	N/A	N/A	vdd 1.5v, core voltage	VDD150_PM_A	None	
	vdd330	4	I	Power	N/A	N/A	N/A	vdd 3.3v, IO voltage	VDD330_PM_A	None	
Group 15	vdd/gnd	11	I	Power	N/A	N/A	N/A	Power pin fill, GND.Vdd1	GND_PM_A / VDD150_P	None	

								.5,Vdd3.3	M_A/		
								as required	VDD330_P		
									M_A		
Total Pins		100									

Please note that pages 549 to 554 are intentionally missing.

BILITHIC PRINTHEADS

1   Background

Silverbrook's bilithic Memjet™ printheads are the target printheads for printing systems which will be controlled by SoPEC and MoPEC devices.

This document presents the format and structure of these printheads, and describes the their possible arrangements in the target systems. It also defines a set of terms used to differentiate between the types of printheads and the systems which use them.

BILITHIC PRINTHEAD CONFIGURATIONS

2   Definitions

This document presents terminology and definitions used to describe the bilithic printhead systems. These terms and definitions are as follows:

- Printhead Type - There are 3 parameters which define the type of printhead used in a system:
- Direction of the data flow through the printhead (clockwise or anti-clockwise, with the printhead shooting ink down onto the page).
- Location of the left-most dot (upper row or lower row, with respect to V_+).
- Printhead footprint (type A or type B, characterized by the data pin being on the left or the right of V_+, where V_+ is at the top of the printhead).
- Printhead Arrangement - Even though there are 8 printhead types, each arrangement has to use a specific pairing of printheads, as discussed in Section 3. This gives 4 pairs of printheads. However, because the paper can flow in either direction with respect to the printheads, there are a total of eight possible arrangements, e.g. Arrangement 1 has a Type 0 printhead on the left with respect to the paper flow, and a Type 1 printhead on the right. Arrangement 2 uses the same printhead pair as Arrangement 1, but the paper flows in the opposite direction.
- Color 0 is always the first color plane encountered by the paper.
- Dot 0 is defined as the nozzle which can print a dot in the left-most side of the page.
- The Even Plane of a color corresponds to the row of nozzles that prints dot 0.

Note that in all of the relevant drawings, printheads should be interpreted as shooting ink down onto the page.

Figure 295 shows the 8 different possible printhead types. Type 0 is identical to the Right Printhead presented in Figure 297 in [1], and Type 1 is the same as the Left Printhead as defined in [1].

While the printheads shown in Figure 295 look to be of equal width (having the same number of nozzles) it is important to remember that in a typical system, a pair of unequal sized printheads may be used.

2.1   COMBINING BILITHIC PRINTHEADS

5   Although the printheads can be physically joined in the manner shown in Figure 296, it is preferable to provide an arrangment that allows greater spacing between the 2 printheads will be required for two main reasons:

- inaccuracies in the backetch
- cheaper manufacturing cost due to decreasing the tolerance requirements in sealing the ink
10   reservoirs behind the printhead

Failing to account for these inaccuracies and tolerances can lead to misalignment of the nozzle rows both vertically and horizontally, as shown in Figure 297.

An even row of color n on printhead A may be vertically misaligned from the even row of color n on
15   printhead B by some number of dots e.g. in Figure 297 this is shown to be 5 dots. And there can also be horizontal misalignment, in that the even row of color n printhead A is not necessarily aligned with the even row of color n+1 on printhead A, e.g. in Figure 297 this horizontal misalignment is 6 dots.

20   The resultant conceptual printhead definition, shown in Figure 297 has properties that are appropriately parameterized in SoPEC and MoPEC to cater for this class of printheads.

The preferred printheads can be characterized by the following features:

- All nozzle rows are the same length (although may be horizontally displaced some number of
25   dots even within a color on a single printhead)
- The nozzles for color n printhead A may not be printing on the same line of the page as the nozzles for color n printhead B. In the example shown in Figure 297, there is a 5 dot displacement between adjacent rows of the printheads.
- The exact shape of the join is an arbitrary shape although is most likely to be sloping (if
30   sloping, it could be sloping either direction)
- The maximum slope is 2 dots per row of nozzles
- Although shift registers are provided in the printhead at the 2 sides of the joined printhead, they do not drive nozzles - this means the printable area is less than the actual shift registers, as highlighted by Figure 298.

35

2.2 Printhead Arrangements

Table 218 defines the printhead pairing and location of the each printhead type, with respect to the flow of paper, for the 8 possible arrangements

Printhead Arrangement	Printhead on left side, with respect to the flow of paper	Printhead on right side, with respect to the flow of paper
Arrangement 1	Type 0	Type 1
Arrangement 2	Type 1	Type 0
Arrangement 3	Type 2	Type 3
Arrangement 4	Type 3	Type 2
Arrangement 5	Type 4	Type 5
Arrangement 6	Type 5	Type 4
Arrangement 7	Type 6	Type 7
Arrangement 8	Type 7	Type 6

5

3 Bilithic Printhead Systems

When using the bilithic printheads, the position of the power/gnd bars coupled with the physical footprint of the printheads mean that we must use a specific pairing of printheads together for printing on the same side of an A4 (or wider) page, e.g. we must always use a Type 0 printhead

10 with a Type 1 printhead etc.

While a given printing system can use any one of the eight possible arrangements of printheads, this document only presents two of them, Arrangement 1 and Arrangement 2, for purposes of illustration. These two arrangements are discussed in subsequent sections of this document.

15 However, the other 6 possibilities also need to be considered.

The main difference between the two printhead arrangements discussed in this document is the direction of the paper flow. Because of this, the dot data has to be loaded differently in Arrangement 1 compared to Arrangement 2, in order to render the page correctly.

20

3.1 EXAMPLE 1: PRINTHEAD ARRANGEMENT 1

Figure 299 shows an Arrangement 1 printing setup, where the bilithic printheads are arranged as follows:

• The Type 0 printhead is on the left with respect to the direction of the paper flow.

25 • The Type 1 printhead is on the right.

557

Table 219 lists the order in which the dot data needs to be loaded into the above printhead system, to ensure color 0-dot 0 appears on the left side of the printed page.

Table 219. Order in which the even and odd dots are loaded for printhead Arrangement 1

Dot Sense	Type 0 printhead when on the left	Type 1 printhead when on the right
Odd	Loaded second in descending order.	Loaded first in descending order.
Even	Loaded first in ascending order.	Loaded second in ascending order.

Figure 300 shows how the dot data is demultiplexed within the printheads.

Figure 301 and Figure 302 show the way in which the dot data needs to be loaded into the printheads in Arrangement 1, to ensure that color 0-dot 0 appears on the left side of the printed page. Note that no data is transferred to the printheads on the first and last edges of SrClk.

3.2 EXAMPLE 2: PRINTHEAD ARRANGEMENT 2

Figure 303 shows an Arrangement 2 printing setup, where the bilithic printheads are arranged as follows:

• The Type 1 printhead is on the left with respect to the direction of the paper flow.
• The Type 0 printhead is on the right.

Table 220 lists the order in which the dot data needs to be loaded into the above printhead system, to ensure color 0-dot 0 appears on the left side of the printed page.

Table 220. Order in which the even and odd dots are loaded for printhead Arrangement 2

Dot Sense	Type 0 printhead when on the right	Type 1 printhead when on the left
Odd	Loaded first in descending order.	Loaded second in descending order.
Even	Loaded second in ascending order.	Loaded first in ascending order.

Figure 304 shows how the dot data is demultiplexed within the printheads.

Figure 305 and Figure 306 show the way in which the dot data needs to be loaded into the printheads in Arrangement 2, to ensure that color 0-dot 0 appears on the left side of the printed page.

5   Note that no data is transferred to the printheads on the first and last edges of SrClk.

4   Conclusions

Comparing the signalling diagrams for Arrangement 1 with those shown for Arrangement 2, it can be seen that the color/dot sequence output for a printhead type in Arrangement 1 is the reverse of

10   the sequence for same printhead in Arrangement 2 in terms of the order in which the color plane data is output, as well as whether even or odd data is output first. However, the order within a color plane remains the same, i.e. odd descending, even ascending.

From Figure 307 and Table 221, it can be seen that the plane which has to be loaded first (i.e. even

15   or odd) depends on the arrangement. Also, the order in which the dots have to be loaded (e.g. even ascending or descending etc.) is dependent on the arrangement.

As well as having a mechanism to cope with the shape of the join between the printheads, as discussed in Section 2.1, if the device controlling the printheads can re-order the bits according to

20   the following criteria, then it should be able to operate in all the possible printhead arrangements:

* Be able to output the even or odd plane first.
* Be able to output even and odd planes in either ascending or descending order, independently.
* Be able to reverse the sequence in which the color planes of a single dot are output to the

25   printhead.

Table 221. Order in which even and odd dots and planes are loaded into the various printhead arrangements

Printhead Arrangement	Left side of printed page	Right side of printed page
Arrangement 1	Even ascending loaded first Odd descending loaded second	Odd descending loaded first Even ascending loaded second
Arrangement 2	Odd descending loaded first Even ascending loaded second	Even ascending loaded first Odd descending loaded second
Arrangement 3	Odd ascending loaded first	Even descending loaded

	Even descending loaded second	first Odd ascending loaded second
Arrangement 4	Even descending loaded first Odd ascending loaded second	Odd ascending loaded first Even descending loaded second
Arrangement 5	Odd ascending loaded first Even descending loaded second	Even descending loaded first Odd ascending loaded second
Arrangement 6	Even descending loaded first Odd ascending loaded second	Odd ascending loaded first Even descending loaded second
Arrangement 7	Even ascending loaded first Odd descending loaded second	Odd descending loaded first Even ascending loaded second
Arrangement 8	Odd descending loaded first Even ascending loaded second	Even ascending loaded first Odd descending loaded second

CMOS SUPPORT ON BILITHIC PRINTHEAD

1   Basic Requirements

To create a two part printhead, of A4/Letter portrait width to print a page in 2 seconds. Matching Left/Right chips can be of different lengths to make up this length facilitating increased wafer usage. the left and right chips are to be imaged on an 8 inch wafer by "Stitching" reticle images.

The memjet nozzles have a horizontal pitch of 32 um, two rows of nozzles are used for a single colour. These rows have a horizontal offset of 16 um. This gives an effective dot pitch of 16 um, or 62.5 dots per mm, or 1587.5 dots per inch, close enough to market as 1600 dpi.

The first nozzle of the right chip should have a 32 um horizontal offset from the final nozzle of the left chip for the same color row. There is no ink nozzle overlap (of the same colour) scheme employed.

1.1   POWER SUPPLY

Vdd/Vpos and Ground supply is made through 30 um wide pads along the length of the chip using conductive adhesive to bus bar beside the chips. Vdd/Vpos is 3.3 Volts. (12V was considered for Vpos but routing of CMOS Vdd at 3.3V would be a problem over the length of the chips, but this will be revisited).

1.2   MEMS CELLS

The preferred memjet device requires 180nJ of energy to fire, with a pulse of current for 1 usec. Assuming 95% efficiency, this requires a 55 ohm actuator drawing 57.4 mA during this pulse.

1.2.1   ISSUE!!!

For 1 pages per 2 second, or ~300 mm * 62.5 (dots/mm) / 2 sec ~= 10 kHz or 100 usec per line. With 1 usec fire pulse cycle, every 100th nozzle needs to fire at the same time. We have 13824 nozzles across the page, so we fire 138 nozzles at a time.

1.2.2   64um unit cell height

This cell would have 4 line spacing between the odd and even dots, and 8 line spacing between adjacent colours.

1.2.3   80 um unit cell height

This cell would have 5 line spacing between the odd and even dots, and 10 line spacing between adjacent colours.

1.3   Versions

1.3.1   6 Colour 1600 dpi with 64 um unit cell

Left and Right Chip.

5   1.3.2   6 Colour 1600 dpi with 80 um unit cell

Left and Right Chip.

1.3.3   4 Colour 800 dpi with 80 um unit cell

For camera application. Single nozzle row per colour.

10

1.4   AIR SUPPLY

Air must be supplied to the MEMS region through holes in the chip.

2   Head Sizes

15   The combined heads have 13824 nozzles per colour totalling 221.184mm of print area. Enough to provide full breadth for A4 (210 mm) and Letter (8.5 inch or 215.9 mm).

Table 1. Head Combinations

Left Head		Right Head	
Stitch Parts	Nozzles per Colour	Stitch Parts	Nozzles per Colour
8	11160	2	2664
7	9744	3	4080
6	8328	4	5496
5	6912	5	6912
4	5496	6	8328
3	4080	7	9744
2	2664	8	11160

20   Nozzles per Colour is calculated as (("Stitch Parts" -1)*118+104)*12. Nozzles per row is half this value. Most likely the 8:2 head set will not be manufactured. The preferred wafer layout, manages to avoid this set, without any loses.

3   Interface

25   Each print head has the same I/O signals (but the Left and Right versions might have a different pin out).

562

Table 2. I/O pins

Name	I/O	Function	Common	Max Speed (MHz)
Data[0-1]	I	Dot data for colours 0 - 5, using Differential Signalling (DataL the complementary signal), colours[0-2] on Data[0], colour[3-5] on Data[1]	No	320
DataL[0-1]	I	complementary signal of Data[0-1]		
SrClk	I	Dot data shift clock using Differential Signalling (SrClkL the complementary signal)	No[1]	320
SrClkL	I	complementary signal of SrClk		
ReadL	I	FrClk, Pr, LSyncL output mode if signal mode bit is set	Yes	1
FrClk	I	Fire pattern shift clock	Yes	1
	O	nozzle test result (mode = 0b001), LsyncL = 0 CMOS testing (mode = 0b111), LsyncL = 1	Yes[2]	
Pr	I	Pulse Profile for all colours	Yes	1[3]
	O	Temperature Output (mode = 0b010), LsyncL = 0 CMOS testing (mode = 0b111), LsyncL = 1	Yes[b]	
LsyncL	I	0 - Capture dot data for next print line	Yes	0.1[4]
	O	CMOS testing (mode = 0b111), LsyncL = 1	Yes[b]	

5   Pins marked as common can be controlled by the same signal from the controller (SOPEC).

3.1   DOT FIRING

To fire a nozzle, three signals are needed. A dot data, a fire signal, and a profile signal. When all signals are high, the nozzle will fire.

10

---

[1]   Functionally could be common, but for timing/electrical reasons should run point to point.

[2]   Can be shared if one side has mode=0b000

[3]   1 MHz cycle, but the resolution of the mark/space ratio may require 50 ns.

[4]   10 kHz cycle, with minimum low pulse of 10 ns (no maximum).

The dot data is provide to the chip through a dot shift register with input *Data[x]*, and clocked into the chip with *SrClk*. The dot data is multiplex on to the *Data* signals, as *Dot[0-2]* on *Data[0]*, and *Dot[3-5]* on *Data[2]*. After the dots are shifted into the dot shift register, this data is transfer into the dot latch, with a low pulse in *LsyncL*. The value in the dot latch forms the dot data used to fire the

5 nozzle. The use of the dot latch allows the next line of data to be loaded into the dot shift register, at the same time the dot pattern in the dot latch is been fired.

Across the top of a column of nozzles, containing 12 nozzles, 2 of each colour (odd and even dots, 4 or 5 lines apart), is two fire register bits and a select register bit. The fire registers forms the fire

10 shift register that runs length of the chip and back again with one register bit in each direction flow. The select register forms the Select Shift Register that runs the length of the chip. The select register, selects which of the two fire registers is used to enables this column. A '0' in this register selects the forward direction fire register, and a '1' selects the reverse direction fire register. This output of this block provides the fire signal for the column.

15

The third signal needed, the profile, is provide for all colours with input *Pr* across the whole colour row at the same time (with a slight propagation delay per column).

3.2 DOT SHIFT REGISTER ORIENTATION

20 The left side print head (chip) and the right side print head that form complete bi-lithic print head, have different nozzle arrangement with respect to the dot order mapping of the dot shift register to the dot position on the page.

With this mapping, the following data streams will need to provided.

Left Head			Right Head	
Size	n-m	dot order	m	
7:3	97 44	[13822,13820,13818,...,4084,4082,4080,] line y+5 [4081,4083,4085,...,13819,13821,13823] line y	40 80	[1,3,5,...,4075,4077,4079,] line y [4078,4076,4074,...,4,2,0] line y+5
6:4	83 28	[13822,13820,13818,...,5500,5498,5496,] line y+5 [5497,5499,5501,...,13819,13821,13823] line y	54 96	[1,3,5,...,5491,5493,5495,] line y [5494,5492,5490,...,4,2,0] line y+5
5:5	69 12	[13822,13820,13818,...,6916,6914,6912,] line y+5 [6913,6915,6917,...,13819,13821,13823] line y	69 12	[1,3,5,...,6907,6909,6911,] line y [6910,6908,6906,...,4,2,0] line y+5
4:6	54 96	[13822,13820,13818,...,8332,8330,8328,] line y+5 [8329,8331,8333,...,13819,13821,13823] line y	83 28	[1,3,5,...,8323,8325,8327,] line y [8326,8324,8322,...,4,2,0] line y+5
3:7	40 80	[13822,13820,13818,...,9748,9746,9744,] line y+5 [9745,97447,9749,...,13819,13821,13823] line y	97 44	[1,3,5,...,9739,9741,9743,] line y 9742,9740,9738,...,4,2,0] line y+5

The data needs to be multiplexed onto the data pins, such that Data[0] has {(C0, C1, C2), (C0, C1, C2)....} in the above order, and Data[1] has {(C3, C4, C5), (C3, C4, C5)....}.

Figure 311 shows the timing of data transfer during normal printing mode. Note *SrClk* has a default state of high and data is transferred on both edges of *SrClk*. If there are *L* nozzles per colour, *SrClk* would have *L+2* edges, where the first and last edges do not transfer data.

Data requires a setup and hold about the both edges of *SrClk*. Data transfers starts on the first rising after *LSyncL* rising. *SrClk* default state is high and needs to return to high after the last data of the line. This means the first edge of *SrClk* (falling) after *LSyncL* rising, and the last edge of *SrClk* as it returns to the default state, no data is transferred to the print head. *LSyncL* rising requires setup to the first falling *SrClk*, and must stay high during the entire line data transfer until after last rising *SrClk*.

3.3   FIRE SHIFT REGISTER

The fire shift register controls the rate of nozzle fire. If the register is full of '1's then the you could print the entire print head in a single *FrClk* cycle, although electrical current limitations will prevent this happening in any reasonable implementation.

Ideally, a '1' is shifted in to the fire shift register, in every n^{th} position, and a '0' in all other position. In this manner, after *n* cycles of *FrClk*, the entire print head will be printed.

The fire shift register and select shift registers allow the generation of a horizontal print line that on close inspection would not have a discontinuity of a "saw tooth" pattern, Figure 312 a) & b) but a "sharks tooth" pattern of c).

This is done by firing 2 nozzles in every 2*n* group of nozzle at the same time starting from the outer 2 nozzles working towards the centre two (or the starting from the centre, and working towards the outer two) at the fire rate controlled by *FrClk*.

To achieve this fire pattern the fire shift register and select shift register need to be set up as show in Figure 313 .

The pattern has shifted a '1' into the fire shift register every n^{th} positions (where *n* is usually is a minimum of about 100) and *n* '1's, followed *n* '0's in the select shift register. At a start of a print cycle, these patterns need to be aligned as above, with the "1000..." of a forward half of fire shift register, matching an *n* grouping of '1' or '0's in the select shift register. As well, with the "1000..." of

a reverse half of the fire shift register, matching an *n* grouping of '1' or '0's in the select shift register. And to continue this print pattern across the butt ends of the chips, the select shift register in each should end with a complete block of n '1's (or '0's).

5   Since the two chips can be of different lengths, initialisation of these patterns is an issue. This is solved by building initialisation circuitry into chips. This circuit is controlled by two registers, nlen(14) and count(14) and b(1). These registers are loaded serially through *Data[0]*, while *LSyncL* is low, and *ReadL* is high with *FrClk*.

10   The scan order from input is b, n[13-0],c[0-13],color[5-0], mode[2-0] therefore b is shifted in last. The system color and mode registers are unrelated to the Fire Shift Register, but are loaded at the same time as this block. There function is described later.

Table 4. Head Combinations Initialisation for *n*=100

Nozzle s L_B	Nozzle s L_A	$nlen_{(A\&B)} =$ n-1	$count_A =$ $(L_A/2) \bmod n$ -1	b_A	b_B	$rem =$ $(L_B/2) \bmod n$	$count_B =$ $(L_A-L_B+rem) \bmod n$ -1
4080	9744	99	71	0	0	40	3
5496	8328	99	63	0	0	48	79
6912	6912	99	55	0	0	56	55

15

The following table shows the values to programme the bi-lithic head pairs using a fire pattern length of 100. The calculation assumes head 'A' is the longest head of the pair and once the registers are initialised with LA FrClk cycles (ReadL='0', LSyncL='1'). *rem* would be the correct value for $count_B$ if chip B was only clocked (FrClk) L_B times. But this chip will be over clocked L_A-L_B

20   cycles. The values of b_A and b_B are either the same or inverse of each other. The actually value does not matter. They need to be different from each other if the select shift registers would end up with different values at the butt ends. If ($L_A/2n$) is even (and $count_A$ is non zero), then the final run in 'A's select shift register will be !b_A. If (L_A-L_B/2) mod *n* is even (and $count_B$ is non zero) then the final run in 'B's select shift register will be !b_B.

25

3.4 SYSTEM REGISTERS

As describe above, the Fire Shift Register generation block, also contains some system registers.

Table 5. System Registers

Name	Size	Function
Color	6	Each bit is an enable for the corresponding colour. If color[X]=0, then Pr_X is 0 and $SrClk_X$ is 0. If color[X]=1, then Pr_X follows the Pr signal and $SrClk_X$ is deserialised SrClk.
Mode	3	Mode[0] = 1, then FrClk pin is used as an output, internally the FrClk signal is set to 0 Mode[1] = 1, then Pr pin is used as an output, internally the Pr signal is set to 0 Mode[2] = 1, then LsyncL pin is used as an output, internally the LsyncL signal is set to 1

5

3.5 PROFILE PATTERN

A profile pattern is repeated at *FrClk* rate. It is expected to be a single pulse about 1us long. But it could be a more complicated series of pulse. The actual pattern depends on the ink type.

The following figure show the external timing to print a line of data. In this example the line is printed

10 in 8 cycles of *FrClk*.

3.6 INTERFACE MODES

The print head has eight different modes controlled by signals *ReadL* and *LSyncL* and system mode register. As seen in Figure 318   with both *LSyncL* and *ReadL* high, the chip in normal

15 printing mode. Some of these modes can operate at the same time, but may interfere with the result of the other modes.

Table 6. Print Head Modes

ReadL	LSyncL	Function	Mode Registe r	Internal Mapping
1	1	Normal Print Mode	000 (XXX)	SrClk=SrClk/3 frclk=FrClk SelClk=0 FsClk=FrClk Scan=0 CoreScan=0

X	0	Dot Load Mode • Dot latches are open, loaded with Dot shift registers, latch once _LSyncL_ returns to 1 (this happens regardless of _ReadL_) • Enables Dot Shift register to capture fire result.	000 (XXX)	
1	0	Fire Load Mode • _Data[0]_ will shift through mode, color, nlen, count and b with _FrClk_	000 (XXX)	SrClk=X frclk=X SelClk=X FsClk=FrClk Scan=1 CoreScan=X
0	1	Reset Nozzle Test • Resets the state of nozzle test circuit	001	SrClk=SrClk FrClk=FrClk SelClk=FrClk FsClk=FrClk Scan=0 CoreScan=1
0	1	CMOS testing mode • The contents of the dot shift registers are serial shifted out on _LsyncL_ (colour0-1), _FrClk_ (colour2-3), _Pr_ (colour4-5) with _SrClk_	111	
0	1	Fire Initialise mode • The contents of the fire shift register and select shift register is generated with _FrClk_	000 (XX0)	
0	0	Temperature Output • The series of Sigma Delta output are clocked out on _Pr_ with _FrClk_. The sum of these bits represent the temperature of the chip.	010	SrClk=X frclk=0 SelClk=0 FsClk=0 Scan=0 CoreScan=X
0	0	Nozzle Test Output • The result of a nozzle test is output on _FrClk_.	001	

3.6.1   Printing

Figure 318 shows show timing for normal printing. During this action, we drop out of *Normal Print Mode*, to *Dot Load Mode* between line transfers. For printing to perform correctly, all other signals should be stable.

5   ### 3.6.2   Initialising for Printing

To initialise for printing the fire shift registers and select shift registers need to be setup into a state as shown in Figure 318 . To do this the chips are put into *Fire Load Mode* and the values for nlen, count and b are serially shifted from Data[0] clocked by *FrClk*. As the two chip have separate Data line, and common FrClk, this happens at the same time. Once this is done, mode is changed to *Fire*

10   *Initialise Mode*, and further L_A *FrClk* cycles are provided to both chips. During all these operation *Pr* should be low, to prevent unintentional firing for nozzles.

3.6.3   Nozzle Testing

Nozzle testing is done by firing a single nozzle at a time and monitoring the *FrClk* pin in the *Nozzle*

15   *Test Output* mode.

Each nozzle has a test switch which closes when the nozzle is fired with an energy level greater than required for normal ink ejection. All 12 switches in a nozzle column are connect in parallel to the following circuit.

20   This circuit is initialised when ever LSyncL is high and ReadL is low (*Reset Nozzle Test* mode). This forces all "switch nodes" to low, and the feedback through lower NOR gate will latches this value. With *LSyncL* low and *ReadL* still low (*Nozzle Test Output* mode) the Testout of the first nozzle column is output on *FrClk*. If any switch is closed, the switch node of this column will be pulled up, and will ripple through to the output as transition from high to low.

25

Nozzle testing requires a setup phase in order to fire only one nozzle. There are many ways to achieve this. Simplest might be to load a single colour with 101010 through the even nozzles, and 010101... for the odd nozzles (0's for all other colours), and set up a fire pattern with $n = L_A/2$. With this fire pattern only one nozzle will fire in each *Pr* pulse. After firing in *Nozzle Test Output* mode, a

30   single *FrClk* will advance to next nozzle, then *Reset* and *Test*. After $L_A/2$ cycles of this testing, a single *SrClk* will advance the dot shift registers to setup the untested nozzles of this colour, and another $L_A/2$ cycles of *FrClk*, *Reset* and *Test* will finished testing this colour. Then repeat test procedure for other colours.

35   ### 3.6.4   Temperature Output

This mode is not well defined yet. In this mode, *Pr* will output a series of ones and zeros clocked by *FrClk*. After a (currently unknown) number of *FrClk* cycles the sum of this series will represent the

temperature of the chip. Clocking frequency in this mode it expected to be in the range 10kHz - 1MHz.

5   The Frequency of *FrClk* and the number of cycles need to be programmable. Since this mode cycles *FrClk*, the result of fire shift register and select shift register would be changed, but in this mode *FrClk* is disabled to these circuit. So printing can resume without reinitialising.

3.6.5   CMOS Testing

CMOS testing is a mode meant for chip testing before MEMS as added to the chip. This mode
10   allows the dot shift register to be shifted out on the *LsyncL,FrClk* and *Pr* pins. Much like the *nozzle test mode*, the nozzles are fired while *LSyncL* is low, but during the firing *SrClk* will be pulsed, loading the dot shift register with the signal that would fire the nozzle. Once captured, the result can be shifted out.

15   The *Dot Load Mode* above violates normal printing procedure by firing the nozzles (*Pr*) and modify the dot shift register (*SrClk*).

4   RETICLE LAYOUT

To make long chips we need to stitch the CMOS (and MEMS) together by overlapping the reticle
20   stepping field. The reticle will contain two areas:

The top edge of *Area 2*, PAD END contains the pads that stitch on bottom edge of *Area 1*, CORE. *Area 1* contains the core array of nozzle logic. The top edge of *Area 1* will stitch to the bottom edge of itself. Finally the bottom edge of *Area 2*, BUTT END will stitch to the top edge of *Area 1*. The
25   BUTT END to used to complete a feedback wiring and seal the chip.

The above region will then be exposed across a wafer bottom to top. *Area 2, Area 1, Area 1*...., *Area 2*. Only the PAD END of *Area 2* needs to fit on the wafer. The final exposure of Area 2 only requires the BUTT END on the wafer.
30

4.1   TSMC U-FRAME REQUIREMENTS.

TSMC will be building us frames 10 mm x 0.23 mm which will be placed either side of both *Area 1* and *Area 2*.

35   TSMC requires 6 mm area for blading between the two exposure area. This translates to 3 mm on the reticle, as some reticules are 2x size, while most are 5x, the worst case must be used.

SECURITY OVERVIEW

1   Introduction

A number of hardware, software and protocol solutions to security issues have been
developed. These range from authorization and encryption protocols for enabling secure
communication between hardware and software modules, to physical and electrical
systems that protect the integrity of integrated circuits and other hardware.

It should be understood that in many cases, principles described with reference to
hardware such as integrated circuits (ie, chips) can be implemented wholly or partly in
software running on, for example, a computer. Mixed systems in which software and
hardware (and combinations) embody various entities, modules and units can also be
constructed using may of these principles, particularly in relation to authorization and
authentication protocols. The particular extent to which the principles described below
can be translated to or from hardware or software will be apparent to one skilled in the
art, and so will not always explicitly be explained.

It should also be understood that many of the techniques disclosed below have
application to many fields other than printing. Some specific examples are described
towards the end of this description.

A "QA Chip" is a quality assurance chip can allows certain security functions and
protocols to be implemented. The preferred QA Chip is described in some detail later in
this specification.

1.5   QA CHIP TERMINOLOGY

The Authentication Protocols documents [5] and [6] refer to QA Chips by their function in particular
protocols:

*   For authenticated reads in [5], ChipR is the QA Chip being read from, and ChipT is the QA
         Chip that identifies whether the data read from ChipR can be trusted. ChipR and ChipT are
         referred to as Untrusted QA Device and Trusted QA Device respectively in [6].

*   For replacement of keys in [5], ChipP is the QA Chip being programmed with the new key,
         and ChipF is the factory QA Chip that generates the message to program the new key. ChipF
         is referred to as the Key Programmer QA Device in [6].

*   For upgrades of data in memory vectors in [5], ChipU is the QA Chip being upgraded, and
         ChipS is the QA Chip that signs the upgrade value. ChipS is referred to as the Value
         Upgrader QA Device and Parameter Upgrader QA Device in [6].

571

Any given physical QA Chip will contain functionality that allows it to operate as an entity in some number of these protocols.

Therefore, wherever the terms ChipR, ChipT, ChipP, ChipF, ChipU and ChipS are used in this document, they are referring to *logical* entities involved in an authentication protocol as defined in [5] and [6].

Physical QA Chips are referred to by their location. For example, each ink cartridge may contain a QA Chip referred to as an INK_QA, with all INK_QA chips being on the same physical bus. In the same way, the QA Chip inside the printer is referred to as PRINTER_QA, and will be on a separate bus to the INK_QA chips.

2   Requirements

2.1   SECURITY

When applied to a printing environment, the functional security requirements for the preferred embodiment are:

- Code of QA chip owner or licensee co-existing safely with code of authorized OEMs
- Chip owner/licensee operating parameters authentication
- Parameters authentication for authorized OEMs
- Ink usage authentication

Each of these is outlined in subsequent sections.

The authentication requirements imply that:

- OEMs and end-users must not be able to replace or tamper with QA chip manufacturer/owner's program code or data
- OEMs and end-users must not be able to perform unauthorized activities for example by calling chip manufacturer/owner's code
- End-users must not be able to replace or tamper with OEM program code or data
- End-users must not be able to call unauthorized functions within OEM program code
- Manufacturer/owner's development program code must not be capable of running on all SoPECs.
- OEMs must be able to test products at their highest upgradable status, yet not be able to ship them outside the terms of their license
- OEMs and end-users must not be able to directly access the print engine pipeline (PEP) hardware, the LSS Master (for QA Chip access) or any other peripheral block with the exception of operating system permitted GPIO pins and timers.

2.1.1 QA Manufacturer/owner code and OEM program code co-existing safely

SoPEC includes a CPU that must run both manufacturer/owner program code and OEM program code. The execution model envisaged for SoPEC is one where Manufacturer/owner program code forms an operating system (O/S), providing services such as controlling the print engine pipeline, interfaces to communications channels etc. The OEM program code must run in a form of user mode, protected from harming the Manufacturer/owner program code. The OEM program code is permitted to obtain services by calling functions in the O/S, and the O/S may also call OEM code at specific times. For example, the OEM program code may request that the O/S call an OEM interrupt service routine when a particular GPIO pin is activated.

In addition, we may wish to permit the OEM code to directly call functions in Manufacturer/owner code with the same permissions as the OEM code. For example, the Manufacturer/owner code may provide SHA1 as a service, and the OEM could call the SHA1 function, but execute that function with OEM permissions and not Silverbook permissions.

A basic requirement then, for SoPEC, is a form of protection management, whereby Manufacturer/owner and OEM program code can co-exist without the OEM program code damaging operations or services provided by the Manufacturer/owner O/S. Since services rely on SoPEC peripherals (such as USB2 Host, LSS Master, Timers etc) access to these peripherals should also be restricted to Manufacturer/owner program code only.

2.1.2 Manufacturer/owner operating parameters authentication

A particular OEM will be licensed to run a Print Engine with a particular set of operating parameters (such as print speed or quality). The OEM and/or end-user can upgrade the operating license for a fee and thereby obtain an upgraded set of operating parameters.

Neither the OEM nor end-user should be able to upgrade the operating parameters without paying the appropriate fee to upgrade the license. Similarly, neither the OEM nor end-user should be able to bypass the authentication mechanism via any program code on SoPEC. This implies that OEMs and end-users must not be able to tamper with or replace Manufacturer/owner program code or data, nor be able to call unauthorized functions within Manufacturer/owner program code.

However, the OEM must be capable of assembly-line testing the Print Engine at the upgraded status before selling the Print Engine to the end-user.

2.1.3 OEM operating parameters authentication

The OEM may provide operating parameters to the end-user independent of the Manufacturer/owner operating parameters. For example, the OEM may want to sell a franking machine[1].

The end-user should not be able to upgrade the operating parameters without paying the appropriate fee to the OEM. Similarly, the end-user should not be able to bypass the authentication mechanism via any program code on SoPEC. This implies that end-users must not be able to tamper with or replace OEM program code or data, as well as not be able to tamper with the PEP blocks or service-related peripherals.

2.2 ACCEPTABLE COMPROMISES

If an end user takes the time and energy to hack the print engine and thereby succeeds in upgrading the single print engine only, yet not be able to use the same keys etc on another print engine, that is an acceptable security compromise. However it doesn't mean we have to make it totally simple or cheap for the end-user to accomplish this.

Software-only attacks are the most dangerous, since they can be transmitted via the internet and have no perceived cost. Physical modification attacks are far less problematic, since most printer users are not likely to want their print engine to be physically modified. This is even more true if the cost of the physical modification is likely to exceed the price of a legitimate upgrade.

2.3 IMPLEMENTATION CONSTRAINTS

Any solution to the requirements detailed in Section 2.1 should also meet certain preferred implementation constraints. These are:

- No flash memory inside SoPEC
- SoPEC must be simple to verify
- Manufacturer/owner program code must be updateable
- OEM program code must be updateable
- Must be bootable from activity on USB2
- Must be bootable from an external ROM to allow stand-alone printer operation
- No extra pins for assigning IDs to slave SoPECs
- Cannot trust the comms channel to the QA Chip in the printer (PRINTER_QA)
- Cannot trust the comms channel to the QA Chip in the ink cartridges (INK_QA)

---

[1] a franking machine prints stamps

- Cannot trust the USB comms channel

These constraints are detailed below.

2.3.1 No flash memory inside SoPEC

The preferred embodiment of SoPEC is intended to be implemented in 0.13 micron or smaller. Flash memory will not be available in any of the target processes being considered.

2.3.2 SoPEC must be simple to verify

All combinatorial logic and embedded program code within SoPEC must be verified before manufacture. Every increase in complexity in either of these increases verification effort and increases risk.

2.3.3 Manufacturer/owner program code must be updateable

It is neither possible nor desirable to write a single complete operating system that is:
- verified completely (see Section 2.3.1)
- correct for all possible future uses of SoPEC systems
- finished in time for SoPEC manufacture

Therefore the complete Manufacturer/owner program code must not *permanently* reside on SoPEC. It must be possible to update the Manufacturer/owner program code as enhancements to functionality are made and bug fixes are applied.

In the worst case, only new printers would receive the new functionality or bug fixes. In the best case, existing SoPEC users can download new embedded code to enable functionality or bug fixes. Ideally, these same users would be obtaining these updates from the OEM website or equivalent, and not require any interaction with Manufacturer/owner.

2.3.4 OEM program code must be updateable

Given that each OEM will be writing specific program code for printers that have not yet been conceived, it is impossible for all OEM program code to be embedded in SoPEC at the ASIC manufacture stage.

Since flash memory is not available (see Section 2.3.1), OEMs cannot store their program code in on-chip flash. While it is theoretically possible to store OEM program code in ROM on SoPEC, this would entail OEM-specific ASICs which would be prohibitively expensive. Therefore OEM program code cannot *permanently* reside on SoPEC.

Since OEM program code must be downloadable for SoPEC to execute, it should therefore be possible to update the OEM program code as enhancements to functionality are made and bug fixes are applied.

5   In the worst case, only new printers would receive the new functionality or bug fixes. In the best case, existing SoPEC users can download new embedded code to enable functionality or bug fixes. Ideally, these same users would be obtaining these updates from the OEM website or equivalent, and not require any interaction with Manufacturer/owner.

10   ### 2.3.5   Must be bootable from activity on USB2
SoPEC can be placed in sleep mode to save power when printing is not required. RAM is not preserved in sleep mode. Therefore any program code and data in RAM will be lost. However, SoPEC must be capable of being woken up by the host when it is time to print again.
In the case of a single SoPEC system, the host communicates with SoPEC via USB2. From
15   SoPEC's point of view, it is activity on the USB2 device port that signals the time to wake up.
In the case of a multi-SoPEC system, the host typically communicates with the Master SoPEC chip (as above), and then the Master relays messages to other Slave SoPECs by sending data out USB2 host port(s) and into the Slave SoPEC's device port. The net result is that the Slave SoPECs and the Master SoPEC all boot as a result of activity on the USB2 device port.
20   Therefore SoPEC must be capable of being woken up by activity on the USB2 device port.

2.3.6   Must be bootable from an external ROM to allow stand-alone printer operation
SoPEC must also support the case where the printer is not connected to a PC (or the PC is currently turned off), and a digital camera or equivalent is plugged into the SoPEC-based printer. In
25   this case, the entire printing application needs to be present within the hardware of the printer. Since the Manufacturer/owner program code and OEM program code will vary depending on the application (see Section 2.3.3 and Section 2.3.4), it is not possible to store the program in SoPEC's ROM.

30   Therefore SoPEC requires a means of booting from a non-PC host. It is possible that this could be accomplished by the OEM adding a USB2-host chip to the printer and simulating the effect of a PC, and thereby download the program code. This solution requires the boot operation to be based on USB2 activity (see Section 2.3.5). However this is an unattractive solution since it adds microprocessor complexity and component cost when only a ROM-equivalent was desired.
35   As a result SoPEC should ideally be able to boot from an external ROM of some kind. *Note that booting from an external ROM means first booting from the internal ROM, and then downloading and authenticating the startup section of the program from the external ROM. This is not the same*

as simply running program code in-situ within an external ROM, since one of the security requirements was that OEMs and end-users must not be able to replace or tamper with Manufacturer/owner program code or data, i.e. we never want to blindly run code from an external ROM.

As an additional point, if SoPEC is in sleep mode, SoPEC must be capable of instigating the boot process due to activity on a programmable GPIO. e.g. a wake-up button. This would be in addition to the standard power-on booting.

2.3.7 No extra pins to assign IDs to slave SoPECs

In a single SoPEC system the host only sends data to the single SoPEC. However in a multi-SoPEC system, each of the slaves needs to be uniquely identifiable in order to be able for the host to send data to the correct slave.

Since there is no flash on board SoPEC (Section 2.3.1) we are unable to store a slave ID in each SoPEC. Moreover, any ROM in each SoPEC will be identical.

It is possible to assign n pins to allow 2^n combinations of IDs for slave SoPECs. However a design goal of SoPEC is to minimize pins for cost reasons, and this is particularly true of features only used in multi-SoPEC systems.

The design constraint requirement is therefore to allow slaves to be IDed via a method that does not require any extra pins. This implies that whatever boot mechanism that satisfies the security requirements of Section 2.1 must also be able to assign IDs to slave SoPECs.

2.3.8 Cannot trust the comms channel to the QA Chip in the printer (PRINTER_QA)

If the printer operating parameters are stored in the non-volatile memory of the Print Engine's on-board PRINTER_QA chip, both Manufacturer/owner and OEM program code cannot rely on the communication channel being secure. It is possible for an attacker to eavesdrop on communications to the PRINTER_QA chip, replace the PRINTER_QA chip and/or subvert the communications channel. It is also possible for this to be true during manufacture of the circuit board containing the SoPEC and the PRINTER_QA chip.

2.3.9 Cannot trust the comms channel to the QA Chip in the ink cartridges (INK_QA)

The amount of ink remaining for a given ink cartridge is stored in the non-volatile memory of that ink cartridge's INK_QA chip. Both Manufacturer/owner and OEM program code cannot rely on the communication channel to the INK_QA being secure. It is possible for an attacker to eavesdrop on

communications to the INK_QA chip, to replace the INK_QA chip and/or to subvert the communications channel. It is also possible for this to be true during manufacture of the consumable containing the INK_QA chip.

5 ### 2.3.10   Cannot trust the inter-SoPEC comms channel (USB2)
In a multi-SoPEC system, or in a single-SoPEC system that has a non-USB2 connection to the host, a given SoPEC will receive its data over a USB2 host port. It is quite possible for an end-user to insert a chip that eavesdrops on and/or subverts the communications channel (for example performs man-in-the-middle attacks).
10

3   Proposed Solution
A proposed solution to the requirements of Section 2, can be summarised as:
- Each SoPEC has a unique id
- CPU with user/supervisor mode
15 - Memory Management Unit
- The unique id is not cached
- Specific entry points in O/S
- Boot procedure, including authentication of program code and operating parameters
- SoPEC physical identification
20

3.1   EACH SoPEC HAS A UNIQUE ID
Each SoPEC needs to contains a unique *SoPEC_id* of minimum size 64-bits. This *SoPEC_id* is used to form a symmetric key unique to each SoPEC: *SoPEC_id_key*. On SoPEC we make use of an additional 112-bit ECID[2] macro that has been programmed with a random number on a per-chip basis.
25 Thus *SoPEC_id* is the 112-bit macro, and the *SoPEC_id_key* is a 160-bit result obtained by SHA1(*SoPEC_id*).

The verification of operating parameters and ink usage depends on *SoPEC_id* being difficult to determine. Difficult to determine means that someone should not be able to determine the id via 30 software, or by viewing the communications between chips on the board. If the *SoPEC_id* is available through running a test procedure on specific test pins on the chip, then depending on the ease by which this can be done, it is likely to be acceptable.

---

[2]Electronic Chip Id

It is important to note that in the proposed solution, compromise of the *SoPEC_id* leads only to compromise of the operating parameters and ink usage on this particular SoPEC. It does not compromise any other SoPEC or all inks or operating parameters in general.

5   It is ideal that the *SoPEC_id* be random, although this is unlikely to occur on standard manufacture processes for ASICs. If the id is within a small range however, it will be able to be broken by brute force. This is why 32-bits is not sufficient protection.

3.2   CPU WITH USER/SUPERVISOR MODE

10   SoPEC contains a CPU with direct hardware support for user and supervisor modes. At present, the intended CPU is the LEON (a 32-bit processor with an instruction set according to the IEEE-1754 standard. The IEEE1754 standard is compatible with the SPARC V8 instruction set).

Manufacturer/owner (operating system) program code will run in supervisor mode, and all OEM
15   program code will run in user mode.

3.3   MEMORY MANAGEMENT UNIT

SoPEC contains a Memory Management Unit (MMU) that limits access to regions of DRAM by defining read, write and execute access permissions for supervisor and user mode. Program code
20   running in user mode is subject to user mode permission settings, and program code running in supervisor mode is subject to supervisor mode settings.

A setting of 1 for a permission bit means that type of access (e.g. read, write, execute) *is* permitted. A setting of 0 for a read permission bit means that that type of access is *not* permitted.
25

At reset and whenever SoPEC wakes up, the settings for all the permission bits are 1 for all supervisor mode accesses, and 0 for all user mode accesses. This means that supervisor mode program code must explicitly set user mode access to be permitted on a section of DRAM.

30   Access permission to all the non-valid address space should be trapped, regardless of user or supervisor mode, and regardless of the access being read, execute, or write.

Access permission to all of the valid non-DRAM address space (for example the PEP blocks) is supervisor read / write access only (no supervisor execute access, and user mode has no acccess
35   at all) with the exception that certain GPIO and Timer registers can also be accessed by user code. These registers will require bitwise access permissions. Each peripheral block will determine how the access is restricted.

With respect to the DRAM and PEP subsystems of SoPEC, typically we would set user read/write/execute mode permissions to be 1/1/0 only in the region of memory that is used for OEM program data, 1/0/1 for regions of OEM program code, and 0/0/0 elsewhere (including the trap table). By contrast we would typically set supervisor mode read/write/execute permissions for this memory to be 1/1/0 (to avoid accidentally executing user code in supervisor mode).

The *SoPEC_id* parameter (see Section 3.1) should only be accessible in supervisor mode, and should only be stored and manipulated in a region of memory that has no user mode access.

3.4   UNIQUE ID IS NOT CACHED
The unique *SoPEC_id* needs to be available to supervisor code and not available to user code. This is taken care of by the MMU (Section 3.3).

However the *SoPEC_id* must also not be accessable via the CPU's data cache or register windows. For example, if the user were to cause an interrupt to occur at a particular point in the program execution when the *SoPEC_id* was being manipulated, it must not be possible for the user program code to turn caching off and then access the *SoPEC_id* inside the data cache. This would bypass any MMU security.

The same must be true of register windows. It must not be possible for user mode program code to read or modify register settings in a supervisor program's register windows.

This means that at the least, the *SoPEC_id* itself must not be cacheable. Likewise, any processed form of the *SoPEC_id* such as the *SoPEC_id_key* (e.g. read into registers or calculated expected results from a QA_Chip) should not be accessable by user program code.

3.5   SPECIFIC ENTRY POINTS IN O/S
Given that user mode program code cannot even call functions in supervisor code space, the question arises as how OEM programs can access functions, or request services. The implementation for this depends on the CPU.

On the LEON processor, the TRAP instruction allows programs to switch between user and supervisor mode in a controlled way. The TRAP switches between user and supervisor register sets, and calls a specific entry point in the supervisor code space in supervisor mode. The TRAP handler dispatches the service request, and then returns to the caller in user mode.

Use of a command dispatcher allows the O/S to provide services that filter access - e.g. a generalised print function will set PEP registers appropriately and ensure QA Chip ink updates occur.

5 The LEON also allows supervisor mode code to call user mode code in user mode. There are a number of ways that this functionality can be implemented. It is possible to call the user code without a trap, but to return to supervisor mode requires a trap (and associated latency).

3.6 BOOT PROCEDURE
10 #### 3.6.1 Basic premise
The intention is to load the Manufacturer/owner and OEM program code into SoPEC's RAM, where it can be subsequently executed. The basic SoPEC therefore, must be capable of downloading program code. However SoPEC must be able to guarantee that only authorized Manufacturer/owner boot programs can be loaded, otherwise anyone could modify the O/S to do
15 anything, and then load that - thereby bypassing the licensed operating parameters.

We perform authentication of program code and data using asymmetric (public-key) digital signatures and *without* using a QA Chip.

20 Assuming we have already downloaded some data and a 160-bit signature into eDRAM, the boot loader needs to perform the following tasks:
- perform SHA-1 on the downloaded data to calculate a digest *localDigest*
- perform asymmetric decryption on the downloaded signature (160-bits) using an asymmetric public key to obtain *authorizedDigest*
25 - If *authorizedDigest* is the PKCS#1 (patent free) form of *localDigest*, then the downloaded data is authorized (the signature must have been signed with the asymmetric private key) and control can then be passed to the downloaded data

Asymmetric decryption is used instead of symmetric decryption because the decrypting key must be held in SoPEC's ROM. If symmetric private keys are used, the ROM can be probed and the security
30 is compromised.

The procedure requires the following data item:
- boot0key = an *n*-bit asymmetric public key

35 The procedure also requires the following two functions:
- SHA-1= a function that performs SHA-1 on a range of memory and returns a 160-bit digest

- decrypt = a function that performs asymmetric decryption of a message using the passed-in key

  - PKCS#1 form of localDigest is 2048-bits formatted as follows: bits 2047-2040 = 0x00, bits 2039-2032 = 0x01, bits 2031-288 = 0xFF..0xFF, bits 287-160 = 0x003021300906052B0E03021A05000414, bits 159-0 = localDigest. For more information, see PKCS#1 v2.1 section 9.2

Assuming that all of these are available (e.g. in the boot ROM), boot loader 0 can be defined as in the following pseudocode:

```
bootloader0(data, sig)
    localDigest ← SHA-1(data)
    authorizedDigest ← decrypt(sig, boot0key)
    expectedDigest                =                0x00|0x01|0xFF..0xFF|
        0x003021300906052B0E03021A05000414    |localDigest)    //
"|" = concat
    If (authorizedDigest == expectedDigest)
        jump to program code at data-start address// will never
return
    Else
        // program code is unauthorized
    EndIf
```

The length of the key will depend on the asymmetric algorithm chosen. The key must provide the equivalent protection of the entire QA Chip system - if the Manufacturer/owner O/S program code can be bypassed, then it is equivalent to the QA Chip keys being compromised. In fact it is worse because it would compromise Manufacturer/owner operating parameters, OEM operating parameters, and ink authentication by software downloaded off the net (e.g. from some hacker).

In the case of RSA, a 2048-bit key is required to match the 160-bit symmetric-key security of the QA Chip. In the case of ECDSA, a key length of 132 bits is likely to suffice. RSA is convenient because the patent (US patent number 4,405,829) expired in September 2000.

There is no advantage to storing multiple keys in SoPEC and having the *external* message choose which key to validate against, because a compromise of any key allows the external user to always select that key.

There is also no particular advantage to having the boot mechanism select the key (e.g. one for USB-based booting and one for external ROM booting) a compromise of the external ROM booting key is enough to compromise all the SoPEC systems.

5   However, there *are* advantages in having multiple keys present in the boot ROM and having a wire-bonding option on the pads select which of the keys is to be used. Ideally, the pads would be connected within the package, and the selection is not available via external means once the die has ben packaged. This means we can have different keys for different application areas (e.g. different uses of the chip), and if any particular SoPEC key is compromised, the die could be kept

10   constant and only the bonding changed. Note that in the worst case of all keys being compromised, it may be economically feasible to change the *boot0key* value in SoPEC's ROM, since this is only a single mask change, and would be easy to verify and characterize.

Therefore the entire security of SoPEC is based on keeping the asymmetric private key paired to
15   *boot0key secure. The entire security of SoPEC is also based on keeping the program that signs*
(i.e. authorizes) datasets using the asymmetric private key paired to boot0key secure.
It may therefore be reasonable to have multiple signatures (and hence multiple signature programs) to reduce the chance of a single point of weakness by a rogue employee. Note that the authentication time increases linearly with the number of signatures, and requires a 2048-bit public
20   key in ROM for each signature.

3.6.2   Hierarchies of authentication
Given that test programs, evaluation programs, and Manufacturer/owner O/S code needs to be written and tested, and OEM program code etc. also needs to be tested, it is not secure to have a
25   single authentication of a monolithic dataset combining Manufacturer/owner O/S, non-O/S, and OEM program code - we certainly don't want OEMs signing Manufacturer/owner program code, and Manufacturer/owner shouldn't have to be involved with the signing of OEM program code.

Therefore we require differing levels of authentication and therefore a number of keys, although the
30   procedure for authentication is identical to the first - a section of program code contains the key and procedure for authenticating the next.

This method allows for any hierarchy of authentication, based on a root key of *boot0key*. For example, assume that we have the following entities:
35   • QACo, Manufacturer/owner's QA/key company. Knows private version of *boot0key*, and owner of security concerns.

- SoPECCo, Manufacturer/owner's SoPEC hardware / software company. Supplies SoPEC ASICs and SoPEC O/S printing software to a ComCo.

- ComCo, a company that assembles Print Engines from SoPECs, Memjet printheads etc, customizing the Print Engine for a given OEM according to a license

5
- OEM, a company that uses a Print Engine to create a printer product to sell to the end-users. The OEM would supply the motor control logic, user interface, and casing.

The levels of authentication hierarchy are as follows:

- QACo writes the boot ROM, agenerates *dataset1*, consisting of a boot loader program that
10 loads and validates *dataset2* and QACo's asymmetric public *boot1key*. QACo signs *dataset0* with the asymmetric private *boot0key*.

- SoPECCo generates *dataset1*, consisting of the print engine security kernel O/S (which incorporates the security-based features of the print engine functionality) and the ComCo's asymmetric public key. Upon a special "formal release" request from SoPECCo, QACo signs
15 *dataset0* with QACo's asymmetric private *boot0key* key. The print engine program code expects to see an operating parameter block signed by the ComCo's asymmetric private key. Note that this is a special "formal release" request to by SoPECCo; the procedure for development versions of the program are described in Section 3.6.3.

- The ComCo generates *dataSet3*, consisting of *dataset1* plus *dataset2*, where *dataset2* is an
20 operating parameter block for a given OEM's print engine licence (according to the print engine license arrangement) signed with the ComCo's asymmetric private key. The operating parameter block (*dataset2*) would contain valid print speed ranges, a *PrintEngineLicenseId*, and the OEM's asymmetric public key. The ComCo can generate as many of these operating parameter blocks for any number of Print Engine Licenses, but cannot write or sign any
25 supervisor O/S program code.

- The OEM would generate *dataset5*, consisting of *dataset3* plus *dataset4*, where *dataset4* is the OEM program code signed with the OEM's asymmetric private key. The OEM can produce as many versions of *dataset5* as it likes (e.g. for testing purposes or for updates to drivers etc) and need not involve Manufacturer/owner, QACo, or ComCo in any way.

30 The relationship is shown below in Figure 325.

When the end-user uses *dataset5*, SoPEC itself validates *dataset1* via the *boot0key* mechanism described in Section 3.6.1. Once *dataset1* is executing, it validates *dataset2*, and uses *dataset2* data to validate *dataset4*. The validation hierarchy is shown in Figure 326.

35

If a key is compromised, it compromises all subsequent authorizations down the hierarchy. In the example from above (and as illustrated in Figure 326) if the OEM's asymmetric private key is

compromised, then O/S program code is not compromised since it is above OEM program code in the authentication hierarchy. However if the ComCo's asymmetric private key is compromised, then the OEM program code is also compromised. A compromise of *boot0key* compromises <u>everything</u> up to SoPEC itself, and would require a mask ROM change in SoPEC to fix.

5

It is worthwhile repeating that in any hierarchy the security of the entire hierarchy is based on keeping the asymmetric private key paired to boot0key secure. It is also a requirement that the program that signs (i.e. authorizes) datasets using the asymmetric private key paired to boot0key secure.

10

3.6.3   Developing Program Code at Manufacturer/owner
The hierarchical boot procedure described in Section 3.6.1 and Section 3.6.2 gives a hierarchy of protection in a final shipped product.

15   It is also desirable to use a hierarchy of protection during software development *within* Manufacturer/owner.

For a program to be downloaded and run on SoPEC during development, it will need to be signed. In addition, we don't want to have to sign each and every Manufacturer/owner development code
20   with the *boot0key*, as it creates the possibility of any developmental (including buggy or rogue) application being run on any SoPEC.

Therefore QACo needs to generate/create a special intermediate boot loader, signed with *boot0key*, that performs the exact same tasks as the normal boot loader, except that it checks the *SoPECid* to
25   see if it is a specific *SoPECid* (or set of *SoPECids*). If the *SoPEC_id* is in the valid set, then the developmental boot loader validates dataset2 by means of its length and a SHA-1 digest of the developmental code[3], and not by a further digital signature. The QACo can give this boot loader to the software development team within Manufacturer/owner. The software team can now write and run any program code, and load the program code using the development boot loader. There is no
30   requirement for the subsequent software program (i.e. the developmental program code) to be signed with any key since the programs can only be run on the particular SoPECs.

---

[3]The SHA-1 digest is to allow the total program load time to simulate the running time of the normal boot loader running on a non-developmental version of the program.

If the developmental boot loader (and/or signature generator) were compromised, or any of the developmental programs were compromised, the worst situation is that an attacker could run programs on that particular set of SoPECs, and on no others.

5   This should greatly reduce the possibility of erroneous programs signed with *boot0key* being available to an attacker (only official releases are signed by *boot0key*), and therefore reduces the possibility of a Manufacturer/owner employee intentionally or inadvertently creating a back door for attackers.

10   The relationship is shown below in Figure 327.

Theoretically the same kind of hierarchy could also be used to allow OEMs to be assured that their program code will only work on specific SoPECs, but this is unlikely to be necessary, and is probably undesirable.

15

3.6.4   Date-limited loaders

It is possible that errors in supervisor program code (e.g. the operating system) could allow attackers to subvert the program in SoPEC and gain supervisor control.

20   To reduce the impact of this kind of attack, it is possible to allocate some bits of the SoPEC_id to form some kind of date. The granularity of the date could be as simple as a single bit that says the date is obtained from the regular IBM ECID, or it could be 6 bits that give 10 years worth of 3-month units.

25   The first step of the program loaded by boot loader 0 could check the SoPEC_id date, and run or refuse to run appropriately. The Manufacturer/owner driver or OS could therefore be limited to run on SoPECs that are manufactured up until a particular date.

This means that the OEM would require a new version of the OS for SoPECs after a particular date,
30   but the new driver could be made to work on all previous versions of SoPEC.

The function simply requires a form of date, whose granularity for working can be determined by agreement with the OEM.

35   For example, suppose that SoPECs are supplied with 3-month granularity in their date components. Manufacturer/owner could ship a version of the OS that works for any SoPEC of the date (i.e. on any chip), or for all SoPECs manufactured during the year etc. The driver issued the next year could

work with all SoPECs up until that years etc. In this way the drivers for a chip will be backwards compatible, but will be deliberately not forwards-compatible. It allows the downloading of a new driver with no problems, but it protects against bugs in one years's driver OS from being used against future SoPECs.

5

Note that the phasing in of a new OS doesn't have to be at the same time as the hardware. For example, the new OS can come in 3 months before the hardware that it supports. However once the new SoPECs are being delivered, the OEM must not ship the older driver with the newer SoPECs, for the old driver will not work on the newer SoPECs. Basically once the OEM has

10   received the new driver, they should use that driver for all SoPEC systems from that point on (old SoPECs will work with the new driver).

This date-limiting feature would most likely be using a field in the ComCo specified operating parameters, so it allows the SoPEC to use date-checking in addition to additional QA Chip related

15   parameter checking (such as the OEM's *PrintEngineLicenseId* etc).

A variant on this theme is a date-window, where a start-date and end-date are specified (as relating to SoPEC manufacture, not date of use).

20   3.6.5   Authenticating operating parameters
Operating parameters need to be considered in terms of Manufacturer/owner operating parameters and OEM operating parameters. Both sets of operating parameters are stored on the PRINTER_QA chip (physically located inside the printer). This allows the printer to maintain parameters regardless of being moved to different computers, or a loss/replacement of host O/S drivers etc.

25

On PRINTER_QA, memory vector M_0 contains the upgradable operating parameters, and memory vectors M_{1+} contains any constant (non-upgradable) operating parameters.

Considering only Manufacturer/owner operating parameters for the moment, there are actually two

30   problems:
a. setting and storing the Manufacturer/owner operating parameters, which should be authorized only by Manufacturer/owner
b. reading the parameters into SoPEC, which is an issue of SoPEC authenticating the data on the PRINTER_QA chip since we don't trust PRINTER_QA.

35   The PRINTER_QA chip therefore contains the following symmetric keys:

- K_0 = *PrintEngineLicense_key*. This key is constant for all SoPECs supplied for a given print engine license agreement between an OEM and a Manufacturer/owner ComCo. K_0 has write permissions to the Manufacturer/owner upgradeable region of M_0 on PRINTER_QA.

- K_1 = *SoPEC_id_key*. This key is unique for each SoPEC (see Section 3.1), and is known only

5   to the SoPEC and PRINTER_QA. K_1 does not have write permissions for anything.

K_0 is used to solve problem (a). It is only used to authenticate the actual upgrades of the operating parameters. Upgrades are performed using the standard upgrade protocol described in [5], with PRINTER_QA acting as the ChipU, and the external upgrader acting as the ChipS.

10

K_1 is used by SoPEC to solve problem (b). It is used to authenticate reads of data (i.e. the operating parameters) from PRINTER_QA. The procedure follows the standard authenticated read protocol described in [5], with PRINTER_QA acting as ChipR, and the embedded supervisor software on SoPEC acting as ChipT. The authenticated read protocol [5] requires the use of a 160-bit nonce,

15   which is a pseudo-random number. This creates the problem of introducing pseudo-randomness into SoPEC that is not readily determinable by OEM programs, especially given that SoPEC boots into a known state. One possibility is to use the same random number generator as in the QA Chip (a 160-bit maximal-lengthed linear feedback shift register) with the seed taken from the value in the *WatchDogTimer* register in SoPEC's timer unit when the first page arrives.

20

Note that the procedure for verifying reads of data from PRINTER_QA does not rely on Manufacturer/owner's key K_0. This means that precisely the same mechanism can be used to read and authenticate the OEM data also stored in PRINTER_QA. Of course this must be done by Manufacturer/owner supervisor code so that *SoPEC_id_key* is not revealed.

25

If the OEM also requires upgradable parameters, we can add an extra key to PRINTER_QA, where that key is an OEM_key and has write permissions to the OEM part of M_0.

In this way, K_1 never needs to be known by anyone except the SoPEC and PRINTER_QA.

30

Each printing SoPEC in a multi-SoPEC system need access to a PRINTER_QA chip that contains the appropriate *SoPEC_id_key* to validate ink useage and operating parameters. This can be accomplished by a separate PRINTER_QA for each SoPEC, or by adding extra keys (multiple *SoPEC_id_keys*) to a single PRINTER_QA.

35

However, if ink usage is not being validated (e.g. if print speed were the only Manufacturer/owner upgradable parameter) then not all SoPECs require access to a PRINTER_QA chip that contains

the appropriate *SoPEC_id_key*. Assuming that OEM program code controls the physical motor speed (different motors per OEM), then the PHI within the first (or only) front-page SoPEC can be programmed to accept (or generate) line sync pulses no faster than a particular rate. If line syncs arrived faster than the particular rate, the PHI would simply print at the slower rate. If the motor

5   speed was hacked to be fast, the print image will appear stretched.

3.6.5.1   Floating operating parameters and dongles

As described in Section 2.1.2, Manufacturer/owner operating parameters include such items as print speed, print quality etc. and are tied to a license provided to an OEM. These parameters are

10   under Manufacturer/owner control. The licensed Manufacturer/owner operating parameters are typically stored in the PRINTER_QA as described in Section 3.6.5.

However there are situations when it is desirable to have a floating upgrade to a license, for use on a printer of the user's choice. For example, OEMs may sell a speed-increase license upgrade that

15   can be plugged into the printer of the user's choice. This form of upgrade can be considered a floating upgrade in that it upgrades whichever printer it is currently plugged into. This dongle is referred to as ADDITIONAL_PRINTER_QA. The software checks for the existence of an ADDITIONAL_PRINTER_QA, and if present the operating parameters are chosen from the values stored on both QA chips.

20

The basic problem of authenticating the additional operating parameters boils down to the problem that we don't trust ADDITIONAL_PRINTER_QA. Therefore we need a system whereby a given SoPEC can perform an authenticated read of the data in ADDITIONAL_PRINTER_QA.

25   We should not write the *SoPEC_id_key* to a key in the ADDITIONAL_PRINTER_QA because:
- then it will be tied specifically to that SoPEC, and the primary intention of the ADDITIONAL_PRINTER_QA is that it be floatable;
- the ink cartridge would then not work in another printer since the other printer would not know the old *SoPEC_id_key* (knowledge of the old key is required in order to change the old key to
30   a new one).
- updating keys is not power-safe (i.e. if at the user's site, power is removed mid-update, the ADDITIONAL_PRINTER_QA could be rendered useless)

The proposed solution is to let ADDITIONAL_PRINTER_QA have two keys:

- K_0 = *FloatingPrintEngineLicense_key*. This key has the same function as the *PrintEngineLicense_key* in the PRINTER_QA[4] in that K_0 has write permissions to the Manufacturer/owner upgradeable region of M_0 on ADDITIONAL_PRINTER_QA.

5 - K_1 = *UseExtParmsLicense_key*. This key is constant for all of the ADDITIONAL_PRINTER_QAs for a given license agreement between an OEM and a Manufacturer/owner ComCo (this is *not* the same key as *PrintEngineLicense_key* which is stored as K_0 in PRINTER_QA). K_1 has no write permissions to anything.

10 K_0 is used to allow writes to the various fields containing operating parameters in the ADDITIONAL_PRINTER_QA. These writes/upgrades are performed using the standard upgrade protocol described in [5], with ADDITIONAL_PRINTER_QA acting as the ChipU, and the external upgrader acting as the ChipS. The upgrader (ChipS) also needs to check the appropriate licensing parameters such as OEM_Id for validity.

15

K_1 is used to allow SoPEC to authenticate reads of the ink remaining and any other ink data. This is accomplished by having the same *UseExtParmsLicense_key* within PRINTER_QA (e.g. in K_2), also with no write permissions. i.e:

- PRINTER_QA.K_2 = *UseExtParmsLicense_key*. This key is constant for all of the
20 PRINTER_QAs for a given license agreement between an OEM and a Manufacturer/owner ComCo. K_2 has no write permissions to anything.

This means there are two shared keys, with PRINTER_QA sharing both, and thereby acting as a bridge between INK_QA and SoPEC.

25 - *UseExtParmsLicense_key* is shared between PRINTER_QA and ADDITIONAL_PRINTER_QA
- *SoPEC_id_key* is shared between SoPEC and PRINTER_QA

All SoPEC has to do is do an authenticated read [6] from ADDITIONAL_PRINTER_QA, pass the
30 data / signature to PRINTER_QA, let PRINTER_QA validate the data / signature, and get PRINTER_QA to produce a similar signature based on the shared *SoPEC_id_key*. It can do so using the *Translate* function [6]. SoPEC can then compare PRINTER_QA's signature with its own calculated signature (i.e. implement a Test function [6] in software on SoPEC), and if the signatures match, the data from ADDITIONAL_PRINTER_QA must be valid, and can therefore be trusted.

---

[4]This can be identical to *PrintEngineLicense_key* in the PRINTER_QA if it is desireable (unlikely) that upgraders can function on PRINTER_QAs as well as ADDITIONAL_PRINTER_QAs

Once the data from ADDITIONAL_PRINTER_QA is known to be trusted, the various operating parameters such as OEM_Id can be checked for validity.

The actual steps of read authentication as performed by SoPEC are:

5
$R_{PRINTER}$ ← PRINTER_QA.random()

R_{DONGLE}, M_{DONGLE}, SIG_{DONGLE} ← DONGLE_QA.read(K1, $R_{PRINTER}$)

R_{SOPEC} ← random()

$R_{PRINTER}$, $SIG_{PRINTER}$ ← PRINTER_QA.translate(K2, R_{DONGLE}, M_{DONGLE}, SIG_{DONGLE}, K1, R_{SOPEC})

10
SIG_{SOPEC} ← HMAC_SHA_1(SoPEC_id_key, M_{DONGLE} | $R_{PRINTER}$ | R_{SOPEC})

If ($SIG_{PRINTER}$ = SIG_{SOPEC})

    // various parms inside M_{DONGLE} (data read from ADDITIONAL_PRINTER_QA) is valid

Else

15
    // the data read from ADDITIONAL_PRINTER_QA is not valid and cannot be trusted

EndIf

3.6.5.2 Dongles tied to a given SoPEC

20
Section 3.6.5.1 describes floating dongles i.e. dongles that can be used on any SoPEC. Sometimes it is desirable to tie a dongle to a specific SoPEC.

Tying a QA_CHIP to be used only on a specific SoPEC can be easily accomplished by writing the PRINTER_QA's chipId (unique serial number) into an appropriate M_0 field on the

25
ADDITIONAL_PRINTER_QA. The system software can detect the match and function appropriately. If there is no match, the software can ignore the data read from the ADDITIONAL_PRINTER_QA.

Although it is also possible to store the *SoPEC_id_key* in one of the keys within the dongle, this

30
must be done in an environment where power will not be removed partway through the key update process (if power is removed during the key update there is a possibility that the dongle QA Chip may be rendered unusable, although this can be checked for after the power failure).

3.6.5.3 OEM assembly-line test

Although an OEM should only be able sell the licensed operating parameters for a given Print Engine, they must be able to assembly-line test[5] or service/test the Print Engine with a different set of operating parameters e.g. a maximally upgraded Print Engine.

5   Several different mechanisms can be employed to allow OEMs to test the upgraded capabilities of the Print Engine. At present it is unclear exactly what kind of assembly-line tests would be performed.

The simplest solution is to use an ADDITIONAL_PRINTER_QA (i..e. special dongle PRINTER_QA
10 as described in Section 3.6.5.1). The ADDITIONAL_PRINTER_QA would contain the operating parameters that maximally upgrade the printer as long as the dongle is connected to the SoPEC. The exact connection may be directly electrical (e.g. via the standard QA Chip connections) or may be over the USB connection to the printer test host depending on the nature of the test. The exact preferred connection is yet to be determined.

15

In the testing environment, the ADDITIONAL_PRINTER_QA also requires a *numberOfImpressions* field inside M_0, which is writeable by K_0. Before the SoPEC prints a page at the higher speed, it decrements the *numberOfImpressions* counter, performs an authenticated read to ensure the count was decremented, and then prints the page. In this way, the total number of pages that can be
20 printed at high speed is reduced in the event of someone stealing the ADDITIONAL_PRINTER_QA device. It also means that multiple test machines can make use of the same ADDITIONAL_PRINTER_QA.

3.6.6   Use of a PrintEngineLicense id

25 Manufacturer/owner O/S program code contains the OEM's asymmetric public key to ensure that the subsequent OEM program code is authentic - i.e. from the OEM. However given that SoPEC only contains a single root key, it is theoretically possible for different OEM's applications to be run identically *physical* Print Engines i.e. printer driver for OEM_1 run on an identically *physical* Print Engine from OEM_2.

30

To guard against this, the Manufacturer/owner O/S program code contains a *PrintEngineLicense_id* code (e.g. 16 bits) that matches the same named value stored as a fixed operating parameter in the PRINTER_QA (i.e. in M_{1+}). As with all other operating parameters, the value of *PrintEngineLicense_id* is stored in PRINTER_QA (and any ADDITIONAL_PRINTER_QA devices)

---

[5]This section is referring to assembly-line testing rather than development testing. An OEM can maximally upgrade a given Print Engine to allow developmental testing of their own OEM program code & mechanics.

at the same time as the other various PRINTER_QA customizations are being applied, before being shipped to the OEM site.

In this way, the OEMs can be sure of differentiating themselves through software functionality.

5

3.6.7   Authentication of ink

The Manufacturer/owner O/S must perform ink authentication [6] during prints. Ink usage authentication makes use of counters in SoPEC that keep an accurate record of the exact number of dots printed for each ink.

10

The ink amount remaining in a given cartridge is stored in that cartridge's INK_QA chip. Other data stored on the INK_QA chip includes ink color, viscosity, Memjet firing pulse profile information, as well as licensing parameters such as OEM_Id, inkType, InkUsageLicense_Id, etc. This information is typically constant, and is therefore likely to be stored in M_{1+} within INK_QA.

15

Just as the Print Engine operating parameters are validated by means of PRINTER_QA, a given Print Engine license may only be permitted to function with specifically licensed ink. Therefore the software on SoPEC could contain a valid set of ink types, colors, OEM_Ids, InkUsageLicense_Ids etc. for subsequent matching against the data in the INK_QA.

20

SoPEC must be able to authenticate reads from the INK_QA, both in terms of ink parameters as well as ink remaining.

To authenticate ink a number of steps must be taken:

25   •   restrict access to dot counts

•   authenticate ink usage and ink parameters via INK_QA and PRINTER_QA

•   broadcast ink dot usage to all SoPECs in a multi-SoPEC system

3.6.7.1   restrict access to dot counts

30   Since the dot counts are accessed via the PHI in the PEP section of SoPEC, access to these registers (and more generally *all* PEP registers) must be only available from supervisor mode, and not by OEM code (running in user mode). Otherwise it might be possible for OEM program code to clear dot counts before authentication has occurred.

35   #### 3.6.7.2   authenticate ink usage and ink parameters via INK_QA and PRINTER_QA

The basic problem of authentication of ink remaining and other ink data boils down to the problem that we don't trust INK_QA. Therefore how can a SoPEC know the initial value of ink (or the ink

parameters), and how can a SoPEC know that after a write to the INK_QA, the count has been correctly decremented.

Taking the first issue, which is determining the initial ink count or the ink parameters, we need a system whereby a given SoPEC can perform an authenticated read of the data in INK_QA.

We cannot write the *SoPEC_id_key* to the INK_QA for two reasons:

- updating keys is not power-safe (i.e. if power is removed mid-update, the INK_QA could be rendered useless)
- the ink cartridge would then not work in another printer since the other printer would not know the old *SoPEC_id_key* (knowledge of the old key is required in order to change the old key to a new one).

The proposed solution is to let INK_QA have two keys:

- K_0 = *SupplyInkLicense_key*. This key is constant for all ink cartridges for a given ink supply agreement between an OEM and a Manufacturer/owner ComCo (this is *not* the same key as *PrintEngineLicense_key* which is stored as K_0 in PRINTER_QA). K_0 has write permissions to the ink remaining regions of M_0 on INK_QA.
- K_1 = *UseInkLicense_key*. This key is constant for all ink cartridges for a given ink usage agreement between an OEM and a Manufacturer/owner ComCo (this is *not* the same key as *PrintEngineLicense_key* which is stored as K_0 in PRINTER_QA). K_1 has no write permissions to anything.

K_0 is used to authenticate the actual upgrades of the amount of ink remaining (e.g. to fill and refill the amount of ink). Upgrades are performed using the standard upgrade protocol described in [5], with INK_QA acting as the ChipU, and the external upgrader acting as the ChipS. The fill and refill upgrader (ChipS) also needs to check the appropriate ink licensing parameters such as OEM_Id, InkType and InkUsageLicense_Id for validity.

K_1 is used to allow SoPEC to authenticate reads of the ink remaining and any other ink data. This is accomplished by having the same *UseInkLicense_key* within PRINTER_QA (e.g. in K_2 or K_3), also with no write permissions.

This means there are two shared keys, with PRINTER_QA sharing both, and thereby acting as a bridge between INK_QA and SoPEC.

- *UseInkLicense_key* is shared between INK_QA and PRINTER_QA
- *SoPEC_id_key* is shared between SoPEC and PRINTER_QA

All SoPEC has to do is do an authenticated read [6] from INK_QA, pass the data / signature to PRINTER_QA, let PRINTER_QA validate the data / signature and get PRINTER_QA to produce a similar signature based on the shared *SoPEC_id_key* (i.e. the *Translate* function [6]). SoPEC can then compare PRINTER_QA's signature with its own calculated signature (i.e. implement a Test function [6] in software on the SoPEC), and if the signatures match, the data from INK_QA must be valid, and can therefore be trusted.

Once the data from INK_QA is known to be trusted, the amount of ink remaining can be checked, and the other ink licensing parameters such as OEM_Id, InkType, InkUsageLicense_Id can be checked for validity.

The actual steps of read authentication as performed by SoPEC are:

```
R_PRINTER  ← PRINTER_QA.random()
R_INK, M_INK, SIG_INK  ← INK_QA.read(K1, R_PRINTER)  //  read  with  key1:
UseInkLicense_key
R_SOPEC  ← random()
R_PRINTER, SIG_PRINTER  ← PRINTER_QA.translate(K2, R_INK, M_INK, SIG_INK, K1,
R_SOPEC)
SIG_SOPEC  ← HMAC_SHA_1(SoPEC_id_key, M_INK | R_PRINTER | R_SOPEC)
If (SIG_PRINTER = SIG_SOPEC)
     // M_INK (data read from INK_QA) is valid
     // M_INK could be ink parameters, such as InkUsageLicense_Id, or
ink remaining
     If (M_INK.inkRemaining = expectedInkRemaining)
         // all is ok
     Else
         // the ink value is not what we wrote, so don't print
anything anymore
     EndIf
Else
     // the data read from INK_QA is not valid and cannot be trusted
EndIf
```

Strictly speaking, we don't need a nonce (R_{SOPEC}) all the time because M_A (containing the ink remaining) should be decrementing between authentications. However we do need one to retrieve the initial amount of ink and the other ink parameters (at power up). This is why taking a random number from the *WatchDogTimer* at the receipt of the first page is acceptable.

In summary, the SoPEC performs the non-authenticated write [6] of ink remaining to the INK_QA chip, and then performs an authenticated read of the data via the PRINTER_QA as per the pseudocode above. If the value is authenticated, _and_ the INK_QA ink-remaining value matches the expected value, the count was correctly decremented and the printing can continue.

5

3.6.7.3 broadcast ink dot usage to all SoPECs in a multi-SoPEC system

In a multi-SoPEC system, each SoPEC _attached to a printhead_ must broadcast its ink usage to all the SoPECs. In this way, each SoPEC will have its own version of the expected ink usage.

10 In the case of a man-in-the-middle attack, at worst the count in a given SoPEC is only its own count (i.e. all broadcasts are turned into 0 ink usage by the man-in-the-middle). We would also require the broadcast amount to be treated as an unsigned integer to prevent negative amounts from being substituted.

15 A single SoPEC performs the update of ink remaining to the INK_QA chip, and then all SoPECs perform an authenticated read of the data via the appropriate PRINTER_QA (the PRINTER_QA that contains their matching _SoPEC_id_key_ - remember that multiple _SoPEC_id_keys_ can be stored in a single PRINTER_QA). If the value is authenticated, and the INK_QA value matches the expected value, the count was correctly decremented and the printing can continue.

20
If any of the broadcasts are not received, or have been tampered with, the updated ink counts will not match. The only case this does not cater for is if each SoPEC is tricked (via a USB2 inter-SoPEC-comms man-in-the-middle attack) into a total that is the same, yet not the true total. Apart from the fact that this is not viable for general pages, at worst this is the maximum amount of ink 25 printed by a single SoPEC. We don't care about protecting against this case.

Since a typical maximum is 4 printing SoPECs, it requires at most 4 authenticated reads. This should be completed within 0.5 seconds, well within the 1-2 seconds/page print time.

30   ### 3.6.8   Example hierarchy
Adding an extra bootloader step to the example from Section 3.6.2, we can break up the contents of program space into logical sections, as shown in Table 227. Note that the ComCo does not provide any program code, merely operating parameters that is used by the O/S.

Table 227. Sections of Program Space

35

section	contents	verifies
0	boot loader 0	section 1 via boot0key

596

(ROM)	SHA-1 function asymmetric decrypt function boot0key	
1	boot loader 1 SoPEC_OS_public_key	section 2 via SoPEC_OS_public_key
2	Manufacturer/owner O/S program code function to generate SoPEC_id_key from SoPEC_id Basic Print Engine ComCo_public_key	section 3 via ComCo_public_key section 4 via OEM_public_key (supplied in section 3) PRINTER_QA data, which includes the PrintEngineLicense_id, Manufacturer/owner operating parameters, and OEM operating parameters (all authenticated via SoPEC_id_key)
3	ComCo license agreement operating parameter ranges, including PrintEngineLicense_id (gets loaded into supervisor mode section of memory) OEM_public_key (gets loaded into supervisor mode section of memory) Any ComCo written user-mode program code (gets loaded into mode mode section of memory)	Is used by section 2 to verify section 4 and range of parameters as found in PRINTER_QA
4	OEM specific program code	OEM operating parameters via calls to Manufacturer/owner O/S code

The verification procedures will be required each time the CPU is woken up, since the RAM is not preserved.

5   3.6.9   What if the CPU is not fast enough?

In the example of Section 3.6.8, every time the CPU is woken up to print a document it needs to perform:

- SHA-1 on all program code and program data
- 4 sets of asymmetric decryption to load the program code and data

- 1 HMAC-SHA1 generation per 512-bits of Manufacturer/owner and OEM printer and ink operating parameters

Although the SHA-1 and HMAC process will be fast enough on the embedded CPU (the program code will be executing from ROM), it may be that the asymmetric decryption will be slow. And this becomes more likely with each extra level of authentication. If this is the case (as is likely), hardware acceleration is required.

A cheap form of hardware acceleration takes advantage of the fact that in most cases the same program is loaded each time, with the first time likely to be at power-up. The hardware acceleration is simply data storage for the *authorizedDigest* which means that the boot procedure now is:

```
slowCPU_bootloader0(data, sig)
    localDigest ← SHA-1(data)
    If (localDigest = previouslyStoredAuthorizedDigest)
        jump to program code at data-start address// will never
return
    Else
        authorizedDigest ← decrypt(sig, boot0key)
        expectedDigest              =              0x00|0x01|0xFF..0xFF|
            0x003021300906052B0E03021A05000414 |localDigest)
        If (authorizedDigest == expectedDigest)
        previouslyStoredAuthorizedDigest ← localDigest
            jump to program code at data-start address// will
never return
        Else
        // program code is unauthorized
    EndIf
```

This procedure means that a reboot of the same authorized program code will only require SHA-1 processing. At power-up, or if new program code is loaded (e.g. an upgrade of a driver over the internet), then the full authorization via asymmetric decryption takes place. This is because the stored digest will not match at power-up and whenever a new program is loaded.

The question is how much preserved space is required.

Each digest requires 160 bits (20 bytes), and this is constant regardless of the asymmetric encryption scheme or the key length. While it is possible to reduce this number of bits, thereby sacrificing security, the cost is small enough to warrant keeping the full digest.

However each level of boot loader requires its own digest to be preserved. This gives a maximum of 20 bytes per loader. Digests for operating parameters and ink levels may also be preserved in the same way, although these authentications should be fast enough not to require cached storage.

5   Assuming SoPEC provides for 12 digests (to be generous), this is a total of 240 bytes. These 240 bytes could easily be stored as 60 × 32-bit registers, or probably more conveniently as a small amount of RAM (eg 0.25 - 1 Kbyte). Providing something like 1 Kbyte of RAM has the advantage of allowing the CPU to store other useful data, although this is not a requirement.

10   In general, it is useful for the boot ROM to know whether it is being started up due to power-on reset, GPIO activity, or activity on the USB2. In the former case, it can ignore the previously stored values (either 0 for registers or garbage for RAM). In the latter cases, it can use the previously stored values. Even without this, a startup value of 0 (or garbage) means the digest won't match and therefore the authentication will occur implictly.

15

3.7   SoPEC PHSYICAL IDENTIFICATION

There must be a mapping of logical to physical since specific SoPECs are responsible for printing on particular physical parts of the page, and/or have particular devices attached to specific pins.

20   The identification process is mostly solved by general USB2 enumeration.

Each slave SoPEC will need to verify the boot broadcast messages received over USB2, and only execute the code if the signatures are valid. Several levels of authorization may occur. However, at some stage, this common program code (broadcast to all of the slave SoPECs and signed by the

25   appropriate asymmetric private key) can, among other things, set the slave SoPEC's id relating to the physical location. If there is only 1 slave, the id is easy to determine, but if there is more than 1 slave, the id must be determined in some fashion. For example, physical location/id determination may be:

- given by the physical USB2 port on the master

30   - related to the physical wiring up of the USB2 interconnects
- based on GPIO wiring. On other systems, a particular physical arrangement of SoPECs may exist such that each slave SoPEC will have a different set of connections on GPIOs. For example, one SoPEC maybe in charge of motor control, while another may be driving the LEDs etc. The unused GPIO pins (not necessarily the same on each SoPEC) can be set as

35   inputs and then tied to 0 or 1. As long as the connection settings are mutually exclusive, program code can determine which is which, and the id appropriately set.

This scheme of slave SoPEC identification does not introduce a security breach. If an attacker rewires the pinouts to confuse identification, at best it will simply cause strange printouts (e.g. swapping of printout data) to occur, while at worst the Print Engine will simply not function.

5   3.8   SETTING UP QA CHIP KEYS

In use, each INK_QA chip needs the following keys:

- K_0 = *SupplyInkLicense_key*
- K_1 = *UseInkLicense_key*

10   Each PRINTER_QA chip tied to a specific SoPEC requires the following keys:

- K_0 = *PrintEngineLicense_key*
- K_1 = *SoPEC_id_key*
- K_2 = *UseExtParmsLicense_key*
- K_3 = *UseInkLicense_key*

15   Note that there may be more than one K_1 depending on the number of PRINTER_QA chips and SoPECs in a system. These keys need to be appropriately set up in the QA Chips before they will function correctly together.

3.8.1   Original QA Chips as received by a ComCo

20   When original QA Chips are shipped from QACo to a specific ComCo their keys are as follows:

- K_0 = *QACo_ComCo_Key0*
- K_1 = *QACo_ComCo_Key1*
- K_2 = *QACo_ComCo_Key2*
- K_3 = *QACo_ComCo_Key3*

25   All 4 keys are only known to QACo. Note that these keys are different for each QA Chip.

3.8.2   Steps at the ComCo

The ComCo is responsible for making Print Engines out of Memjet printheads, QA Chips, PECs or SoPECs, PCBs etc.

30

In addition, the ComCo must customize the INK_QA chips and PRINTER_QA chip on-board the print engine before shipping to the OEM.

There are two stages:

- replacing the keys in QA Chips with specific keys for the application (i.e. INK_QA and
35   PRINTER_QA)
- setting operating parameters as per the license with the OEM

600

3.8.2.1 Replacing keys

The ComCo is issued QID hardware [4] by QACo that allows programming of the various keys (except for K_1) in a given QA Chip to the final values, following the standard ChipF/ChipP replace key (indirect version) protocol [6]. The indirect version of the protocol allows each

5   QACo_ComCo_Key to be different for each SoPEC.

In the case of programming of PRINTER_QA's K_1 to be SoPEC_id_key, there is the additional step of transferring an asymmetrically encrypted SoPEC_id_key (by the public-key) along with the nonce (R_P) used in the replace key protocol to the device that is functioning as a ChipF. The ChipF must

10   decrypt the SoPEC_id_key so it can generate the standard replace key message for PRINTER_QA (functioning as a ChipP in the ChipF/ChipP protocol). The asymmetric key pair held in the ChipF equivalent should be unique to a ComCo (but still known only by QACo) to prevent damage in the case of a compromise.

15   Note that the various keys installed in the QA Chips (both INK_QA and PRINTER_QA) are only known to the QACo. The OEM only uses QIDs and QACo supplied ChipFs. The replace key protocol [6] allows the programming to occur without compromising the old or new key.

3.8.2.2 Setting operating parameters

20   There are two sets of operating parameters stored in PRINTER_QA and INK_QA:
*   fixed
*   upgradable

The fixed operating parameters can be written to by means of a non-authenticated writes [6] to M_{1+} via a QID [4], and permission bits set such that they are ReadOnly.

25

The upgradable operating parameters can only be written to after the QA Chips have been programmed with the correct keys as per Section 3.8.2.1. Once they contain the correct keys they can be programmed with appropriate operating parameters by means of a QID and an appropriate ChipS (containing matching keys).

AUTHENTICATION PROTOCOLS

1   Introduction

The following describes authentication protocols for general authentication applications, but with specific reference to the QA Chip.

5

The intention is to show the broad form of possible protocols for use in different authentication situations, and can be used as a reference when subsequently defining an implementation specification for a particular application. As mentioned earlier, although the protocols are described in relation to a printing environment, many of them have wider application such as, but not limited to, those described at the end of this specification.

10

2   Nomenclature

The following symbolic nomenclature is used throughout this document:

Table 228. Summary of symbolic nomenclature

15

Symbol	Description
F[X]	Function F, taking a single parameter X
F[X, Y]	Function F, taking two parameters, X and Y
X | Y	X concatenated with Y
X ∧ Y	Bitwise X AND Y
X ∨ Y	Bitwise X OR Y (inclusive-OR)
X ⊕ Y	Bitwise X XOR Y (exclusive-OR)
¬X	Bitwise NOT X (complement)
X ← Y	X is assigned the value Y
X ← {Y, Z}	The domain of assignment inputs to X is Y and Z
X = Y	X is equal to Y
X ≠ Y	X is not equal to Y
⇓X	Decrement X by 1 (floor 0)
⇑X	Increment X by 1 (modulo register length)
Erase X	Erase Flash memory register X
SetBits[X, Y]	Set the bits of the Flash memory register X based on Y
Z ← ShiftRight[X, Y]	Shift register X right one bit position, taking input bit from Y and placing the output bit in Z

3   PSEUDOCODE

3.1   Asynchronous

The following pseudocode:

```
var = expression
```

means the var signal or output is equal to the evaluation of the expression.

3.2   Synchronous

The following pseudocode:

```
var ← expression
```

means the var register is assigned the result of evaluating the expression during this cycle.

3.3   Expression

Expressions are defined using the nomenclature in Table 228 above. Therefore:

```
var = (a = b)
```

is interpreted as the var signal is 1 if a is equal to b, and 0 otherwise.

4.   Intentionally blank

5   Basic Protocols

5.1   PROTOCOL BACKGROUND

This protocol set is a restricted form of a more general case of a multiple key single memory vector protocol. It is a restricted form in that the memory vector M has been optimized for Flash memory utilization:

- M is broken into multiple memory vectors (semi-fixed and variable components) for the purposes of optimizing flash memory utilization. Typically M contains some parts that are fixed at some stage of the manufacturing process (eg a batch number, serial number etc.), and once set, are not ever updated. This information does not contain the amount of consumable remaining, and therefore is not read or written to with any great frequency.

- We therefore define M_0 to be the M that contains the frequently updated sections, and the remaining Ms to be rarely written to. Authenticated writes only write to M_0, and non-authenticated writes can be directed to a specific M_n. This reduces the size of permissions that are stored in the QA Chip (since key-based writes are not required for Ms other than M_0). It also means that M_0 and the remaining Ms can be manipulated in different ways, thereby increasing flash memory longevity.

5.2   REQUIREMENTS OF PROTOCOL

Each QA Chip contains the following values:

N   The maximum number of keys known to the chip.

T	The number of vectors M is broken into.
K_N	Array of N secret keys used for calculating $F_{Kn}[X]$ where K_n is the nth element of the array.
R	Current random number used to ensure time varying messages. Each chip instance must be seeded with a different initial value. Changes for each signature generation.
M_T	Array of T memory vectors. Only M_0 can be written to with an authorized write, while all Ms can be written to in an unauthorized write. Writes to M_0 are optimized for Flash usage, while updates to any other M_{1+} are expensive with regards to Flash utilization, and are expected to be only performed once per section of M_n. M_1 contains T, N and f in ReadOnly form so users of the chip can know these two values.
P_{T+N}	T+N element array of access permissions for each part of M. Entries n={0... T-1} hold access permissions for non-authenticated writes to M_n (no key required). Entries n={T to T+N-1}hold access permissions for authenticated writes to M_0 for K_n. Permission choices for each part of M are Read Only, Read/Write, and Decrement Only.
C	3 constants used for generating signatures. C_1, C_2, and C_3 are constants that pad out a sub-message to a hashing boundary, and all 3 must be different.

Each QA Chip contains the following private function:

$S_{Kn}[N,X]$	*Internal function only.* Returns $S_{Kn}[X]$, the result of applying a digital signature function S to X based upon the appropriate key K_n. The digital signature must be long enough to counter the chances of someone generating a random signature. The length depends on the signature scheme chosen, although the scheme chosen for the QA Chip is HMAC-SHA1, and therefore the length of the signature is 160 bits.

Additional functions are required in certain QA Chips, but these are described as required.

5.3   READ PROTOCOLS

The set of read protocols describe the means by which a System reads a specific data vector M_t from a QA Chip referred to as *ChipR*.

We assume that the communications link to ChipR (and therefore ChipR itself) is not trusted. If it were trusted, the System could simply read the data and there is no issue. Since the communications link to ChipR is not trusted and ChipR cannot be trusted, the System needs a way of authenticating the data as actually being from a real ChipR.

Since the read protocol must be capable of being implemented in physical QA Chips, we cannot use asymmetric cryptography (for example the ChipR signs the data with a private key, and System validates the signature using a public key).

This document describes two read protocols:

- direct validation of reads
- indirect validation of reads.

5.3.1   Direct Validation of Reads

In a direct validation read protocol we require two QA Chips: *ChipR* is the QA Chip being read, and *ChipT* is the QA Chip we entrust to tell us whether or not the data read from ChipR is trustworthy. The basic idea is that system asks ChipR for data, and ChipR responds with the data and a signature based on a secret key. System then asks ChipT whether the signature supplied by ChipR is correct. If ChipT responds that it is, then System can trust that data just read from ChipR. Every time data is read from ChipR, the validation procedure must be carried out.

Direct validation requires the System to trust the communication line to ChipT. This could be because ChipT is in physical proximity to the System, and both System and ChipT are in a trusted (e.g. Silverbrook secure) environment. However, since we need to validate the read, ChipR by definition must be in a non-trusted environment.

Each QA Chip protects its signature generation or verification mechanism by the use of a nonce.

The protocol requires the following publicly available functions in ChipT:

Random[]   Returns R (does not advance R).

Test[n,X, Y, Z]   Advances R and returns 1 if $S_{Kn}[R|X|C_1|Y] = Z$. Otherwise returns 0. The time taken to calculate and compare signatures must be independent of data content.

The protocol requires the following publicly available functions in ChipR:

Read[n, t, X]   Advances R, and returns R, M_t, $S_{Kn}[X|R|C_1|M_t]$. The time taken to calculate the signature must not be based on the contents of X, R, M_t, or K. If t is invalid, the function assumes t=0.

To read ChipR's memory M_t in a validated way, System performs the following tasks:

a. System calls ChipT's Random function;

b. ChipT returns R_T to System;

c. System calls ChipR's Read function, passing in some key number *n1*, the desired data vector number *t*, and R_T (from b);

d. ChipR updates R_R, then calculates and returns R_R, M_{Rt}, $S_{Kn1}[R_T|R_R|C_1|M_{Rt}]$;

e. System calls ChipT's Test function, passing in the key to use for signature verification *n2*, and the results from d (i.e. R_R, M_{Rt}, $S_{Kn1}[R_T|R_R|C_1|M_{Rt}]$);

f. System checks response from ChipT. If the response is 1, then the M_t read from ChipR is considered to be valid. If 0, then the M_t read from ChipR is considered to be invalid.

The choice of n1 and n2 must be such that ChipR's K_{n1} = ChipT's K_{n2}.

The data flow for this read protocol is shown in Figure 328.

From the System's perspective, the protocol would take on a form like the following pseudocode:

```
RT ← ChipT.Random()

RR, MR, SIGR ← ChipR.Read(keyNumOnChipR,desiredM, RT)

ok ← ChipT.Test(keyNumOnChipT, RR, MR, SIGR)

If (ok = 1)

// MR is to be trusted

Else

// MR is not to be trusted

EndIf
```

10   With regards to security, if an attacker finds out ChipR's K_{n1}, they can replace the ChipR by a fake ChipR because they can create signatures. Likewise, if an attacker finds out ChipT's K_{n2}, they can replace the ChipR by a fake ChipR because ChipR's K_{n1} = ChipT's K_{n2}. Moreover, they can use the ChipRs on any system that shares the same key.

15   The only way of restricting exposure due to key reveals is to restrict the number of systems that match ChipR and ChipT. i.e. vary the key as much as possible. The degree to which this can be done will depend on the application. In the case of a PRINTER_QA acting as a ChipT, and an INK_QA acting as a ChipR, the same key must be used on all systems where the particular INK_QA data must be validated.

20

In all cases, ChipR must contain sufficient information to produce a signature. Knowing (or finding out) this information, whatever form it is in, allows clone ChipRs to be built.

5.3.2   Indirect Validation of Reads

25   In a direct validation protocol (see Section 5.3.1), the System validates the correctness of data read from ChipR by means of a trusted chip ChipT. This is possible because ChipR and ChipT share some secret information.

However, it is possible to extend trust via indirect validation. This is required when we trust ChipT,

30   but ChipT doesn't know how to validate data from ChipR. Instead, ChipT knows how to validate data from *ChipI* (some intermediate chip) which in turn knows how to validate data from either another ChipI (and so on up a chain) or ChipR. Thus we have a chain of validation.

The means of validation chains is translation of signatures. $ChipI_n$ translates signatures from higher

35   up the chain (either $ChipI_{n-1}$ or from ChipR at the start of the chain) into signatures capable of being passed to the next stage in the chain (either $ChipI_{n+1}$ or to ChipT at the end of the chain). A given

ChipI can only translate signatures if it knows the key of the previous stage in the chain as well as the key of the next stage in the chain.

The protocol requires the following publicly available functions in ChipI:

5 Random[]   Returns R (does not advance R).

Translate[n1,X, Y, Z,n2,A]   Returns 1, $S_{Kn2}[A|R|C_1|Y]$ and advances R if $Z = S_{Kn1}[R|X|C_1|Y]$.
                              Otherwise returns 0, 0. The time taken to calculate and compare
                              signatures must be independent of data content.

10 The data flow for this signature translation protocol is shown in Figure 329:

Note that R_{prev} is eventually R_R, and R_{next} is eventually R_T. In the multiple ChipI case, R_{prev} is the R_I of $ChipI_{n-1}$ and R_{next} is R_I of $ChipI_{n+1}$. The R_{prev} of the first ChipI in the chain is R_R, and the R_{next} of the last ChipI in the chain is R_T.

15

Assuming at least 1 ChipT, the System would need to perform the following tasks in order to read ChipR's memory M_t in an indirectly validated way:

a. System calls $ChipI_n$'s Random function;

b. $ChipI_0$ returns R_{I0} to System;

20 c. System calls ChipR's Read function, passing in some key number $n0$, the desired data vector number t, and R_{I0} (from b);

d. ChipR updates R_R, then calculates and returns R_R, M_{Rt}, $S_{Kn0}[R_{In}|R_R|C_1|M_{Rt}]$;

e. System assigns R_R to R_{prev} and $S_{Kn0}[R_{In}|R_R|C_1|M_{Rt}]$ to SIG_{prev}

f. System calls the next-chip-in-the-chain's Random function (either $ChipI_{n+1}$ or ChipT)

25 g. The next-chip-in-the-chain will return R_{next} to System

h. System calls $ChipI_n$'s Translate function, passing in $n1_n$ (translation input key number), R_{prev}, M_{Rt}, SIG_{prev}), $n2_n$ (translation output key number) and the results from g (R_{next});

i. ChipI returns testResult and SIG_I to System

j. If testResult = 0, then the validation has failed, and the M_t read from ChipR is considered to be
30   invalid. Exit with failure.

k. If the next chip in the chain is a ChipI, assign SIG_I to SIG_{prev} and go to step f

l. System calls ChipT's Test function, passing in n_t, R_{prev}, M_{Rt}, and SIG_{prev};

m. System calls System checks response from ChipT. If the response is 1, then the M_t read from ChipR is considered to be valid. If 0, then the M_t read from ChipR is considered to be invalid.

35

For the Translate function to work, $ChipI_n$ and $ChipI_{n+1}$ must share a key. The choice of n1 and n2 in the protocol described must be such that $ChipI_n$'s $K_{n2} = ChipI_{n+1}$'s K_{n1}.

Note that Translate is essentially a "Test plus resign" function. From an implementation point of view the first part of Translate is identical to Test.

Note that the use of ChipIs and the translate function merely allows signatures to be transformed. At the end of the translation chain (if present) will be a ChipT requiring the use of a Test function. There can be any number of ChipIs in the chain to ChipT as long as the Translate function is used to map signatures between $ChipI_n$ and $ChipI_{n+1}$ and so on until arrival at the final destination (ChipT).

From the System's perspective, a read protocol using at least 1 ChipI would take on a form like the following pseudocode:

```
R_next ← ChipI[0].Random()
R_prev, M_R, SIG_prev ← ChipR.Read(keyNumOnChipR, desiredM,
R_next)
ok = 1
i = 0
while ((i < iMax) AND ok)
For i ← 0 to iMax
If (i = iMax)
    R_next ← ChipT.Random()
Else
    R_next ← ChipI[i+1].Random()
EndIf
ok, SIG_prev ← ChipI[i].Translate(iKey[i], R_prev, M_R,
SIG_prev, oKey[i], R_next)
R_prev = R_next
If (ok = 0)
    // M_R is not to be trusted
EndIf
EndFor
ok ← ChipT.Test(keyNumOnChipT, R_prev, M_R, SIG_prev)
If (ok = 1)
// M_R is to be trusted
Else
// M_R is not to be trusted
EndIf
```

5.3.3 Additional Comments on Reads

In the Memjet printing environment, certain implementations will exist where the operating parameters are stored in QA Chips. In this case, the system must read the data from the QA Chip using an appropriate read protocol.

If the connection is trusted (e.g. to a virtual QA Chip in software), a generic Read is sufficient. If the connection is not trusted, it is ideal that the System have a trusted ChipT in the form of software (if possible) or hardware (e.g. a QA Chip on board the same silicon package as the microcontroller and firmware). Whether implemented in software or hardware, the QA Chip should contain an appropriate key that is unique per print engine. Such a key setup would allow reads of print engine parameters and also allow indirect reads of consumables (from a consumable QA Chip).

If the ChipT is physically separate from System (e.g. ChipT is on a board connected to System) System *must also occasionally* (based on system clock for example) call ChipT's Test function with bad data, expecting a 0 response. This is to reduce the possibility of someone inserting a fake ChipT into the system that always returns 1 for the Test function.

5.4 UPGRADE PROTOCOLS

This set of protocols describe the means by which a System upgrades a specific data vector M_t within a QA Chip (*ChipU*). The data vector may contain information about the functioning of the device (e.g. the current maximum operating speed) or the amount of a consumable remaining.

The updating of M_t in ChipU falls into two categories:

- non-authenticated writes, where anyone is able to update the data vector
- authenticated writes, where only authorized entities are able to upgrades data vectors

5.4.1 Non-authenticated writes

This is the most frequent type of write, and takes place between the System / consumable during normal everyday operation for M_0, and during the manufacturing process for M_{1+}.

In this kind of write, the System wants to change M_t within ChipU subject to P. For example, the System could be decrementing the amount of consumable remaining. Although *System does not need to know and of the Ks or even have access to a trusted chip* to perform the write, the System must follow a non-authenticated write by an authenticated read if it needs to know that the write was successful.

The protocol requires ChipU to contain the following publicly available function:

Write[t, X]   Writes X over those parts of M_t subject to P_t and the existing value for M.

              To authenticate a write of M_{new} to ChipA's memory M:

a. System calls ChipU's Write function, passing in M_{new};

b. The authentication procedure for a Read is carried out (see Section 5.3 on page 604);

c. If the read succeeds in such a way that M_{new} = M returned in b, the write succeeded. If not, it failed.

Note that if these parameters are transmitted over an error-prone communications line (as opposed to internally or using an additional error-free transport layer), then an additional checksum would be required to prevent the wrong M from being updated or to prevent the correct M from being updated to the wrong value. For example, SHA-1[t,X] should be additionally transferred across the communications line and checked (either by a wrapper function around Write or in a variant of Write that takes a hash as an extra parameter).

This is the most frequent type of write, and takes place between the System / consumable during normal everyday operation for M_0, and during the manufacturing process for M_{1+}.

5.4.2   Authenticated writes

In the QA Chip protocols, M_0 is defined to be the only data vector that can be upgraded in an authenticated way. This decision was made primarily to simplify flash management, although it also helps to reduce the permissions storage requirements.

In this kind of write, System wants to change Chip U's M_0 in an authorized way, without being subject to the permissions that apply during normal operation. For example, a consumable may be at a refilling station and the normally Decrement Only section of M_0 should be updated to include the new valid consumable. In this case, the chip whose M_0 is being updated must authenticate the writes being generated by the external System and in addition, apply the appropriate permission for the key to ensure that only the correct parts of M_0 are updated. Having a different permission for each key is required as when multiple keys are involved, all keys should not necessarily be given open access to M_0. For example, suppose M_0 contains printer speed and a counter of money available for franking. A ChipS that updates printer speed should not be capable of updating the amount of money. Since $P_{0...T-1}$ is used for non-authenticated writes, each K_n has a corresponding permission P_{T+n} that determines what can be updated in an authenticated write.

The basic principle of the authenticated write (or upgrade) protocol is that the new value for the M_t must be signed before ChipU accepts it. The QA Chip responsible for generating the signature

610

(ChipS) must first validate that the ChipU is valid by reading the old value for M_t. Once the old value is seen as valid, a new value can be signed by ChipS and the resultant data plus signature passed to ChipU. Note that both chips distrust each other.

5 There are two forms of authenticated writes. The first form is when both ChipU and ChipS directly store the same key. The second is when both ChipU and ChipS store different versions of the key and a transforming procedure is used on the stored key to generate the required key - i.e. the key is indirectly stored. The second form is slightly more complicated, and only has value when the ChipS is not readily available to an attacker.

10

5.4.2.1 Direct authenticated writes

The direct form of the authenticated write protocol is used when the ChipS and ChipU are equally available to an attacker. For example, suppose that ChipU contains a printer's operating speed. Suppose that the speed can be increased by purchasing a ChipS and inserting it into the printer

15 system. In this case, the ChipS and ChipU are equally available to an attacker. This is different from upgrading the printer over the internet where the effective ChipS is in a remote location, and thereby not as readily available to an attacker.

The direct authenticated write protocol requires ChipU to contain the following publicly available

20 functions:

Read[n, t, X]   Advances R, and returns R, M_t, $S_{Kn}[X|R|C_1|M_t]$. The time taken to calculate the signature must not be based on the contents of X, R, M_t, or K.

WriteA[n, X, Y, Z]   Advances R, replaces M_0 by Y subject to P_{T+n}, and returns 1 only if $S_{Kn}[R|X|C_1|Y]$

25   = Z. Otherwise returns 0. The time taken to calculate and compare signatures must be independent of data content. This function is identical to ChipT's Test function except that it additionally writes Y subject to P_{T+n} to its M when the signature matches.

Authenticated writes require that the System has access to a ChipS that is capable of generating

30 appropriate signatures.

In its basic form, ChipS requires the following variables and function:

SignM[n,V,W,X,Y,Z]   Advances R, and returns R, $S_{Kn}[W|R|C_1|Z]$ only if Y = $S_{Kn}[V|W|C_1|X]$.
Otherwise returns all 0s. The time taken to calculate and compare signatures must

35   be independent of data content.

To update ChipU's M vector:

a. System calls ChipU's Read function, passing in n1, 0 (desired vector number) and 0 (the random value, but is a don't-care value) as the input parameters;

b. ChipU produces R_U, M_{U0}, $S_{Kn1}[0|R_U|C_1|M_{U0}]$ and returns these to System;

c. System calls ChipS's SignM function, passing in n2 (the key to be used in ChipS), 0 (the random value as used in a), R_U, M_{U0}, $S_{Kn1}[0|R_U|C_1|M_{U0}]$, and M_D (the desired vector to be written to ChipU);

d. ChipS produces R_S and $S_{Kn2}[R_U|R_S|C_1|M_D]$ if the inputs were valid, and 0 for all outputs if the inputs were not valid.

e. If values returned in d are non zero, then ChipU is considered authentic. System can then call ChipU's WriteA function with these values from d.

f. ChipU should return a 1 to indicate success. A 0 should only be returned if the data generated by ChipS is incorrect (e.g. a transmission error).

The choice of n1 and n2 must be such that ChipU's K_{n1} = ChipS's K_{n2}.

The data flow for authenticated writes is shown in Figure 330.

Note that this protocol allows ChipS to generate a signature for any desired memory vector MD, and therefore a stolen ChipS has the ability to effectively render the particular keys for those parts of M_0 in ChipU irrelevant.

It is therefore not recommended that the basic form of ChipS be ever implemented except in specifically controlled circumstances.

It is much more secure to limit the powers of ChipS. The following list covers some of the variants of limiting the power of ChipS:

a. the ability to upgrade a limited number of times

b. the ability to upgrade based on a credit value - i.e. the upgrade amount is decremented from the local value, and effectively transferred to the upgraded device

c. the ability to upgrade to a fixed value or from a limited list

d. the ability to upgrade to any value

e. the ability to only upgrade certain data fields within M

In many of these variants, the ability to refresh the ChipS in some way (e.g. with a new count or credit value) would be a useful feature.

In certain cases, the variant is in ChipS, while ChipU remains the same. It may also be desirable to create a ChipU variant, for example only allowing ChipU to only be upgraded a specific number of times.

5.4.2.1.1 Variant example

This section details the variant for the ability to upgrade a memory vector to any value a specific number of times, but the upgrade is only allowed to affect certain fields within the memory vector i.e. a combination of (a), (d), and (e) above.

In this example, ChipS requires the following variables and function:

CountRemaining — Part of ChipS's M_0 that contains the number of signatures that ChipS is allowed to generate. Decrements with each successful call to SignM and SignP. Permissions in ChipS's $P_{0..T-1}$ for this part of M_0 needs to be ReadOnly once ChipS has been setup. Therefore CountRemaining can only be updated by another ChipS that will perform updates to that part of M_0 (assuming ChipS's Ps allows that part of M_0 to be updated).

Q — Part of M that contains the write permissions for updating ChipU's M. By adding Q to ChipS we allow different ChipSs that can update different parts of M_U. Permissions in ChipS's $P_{0..T-1}$ for this part of M needs to be ReadOnly once ChipS has been setup. Therefore Q can only be updated by another ChipS that will perform updates to that part of M.

SignM[n,V,W,X,Y,Z] — Advances R, decrements CountRemaining and returns R, Z_{QX} (Z applied to X with permissions Q), $S_{Kn}[W|R|C_1|Z_{QX}]$ only if $Y = S_{Kn}[V|W|C_1|X]$ and CountRemaining > 0. Otherwise returns all 0s. The time taken to calculate and compare signatures must be independent of data content.

To update ChipU's M vector:

a. System calls ChipU's Read function, passing in n1, 0 (desired vector number) and 0 (the random value, but is a don't-care value) as the input parameters;

b. ChipU produces R_U, M_{U0}, $S_{Kn1}[0|R_U|C_1|M_{U0}]$ and returns these to System;

c. System calls ChipS's SignM function, passing in n2 (the key to be used in ChipS), 0 (as used in a), R_U, M_{U0}, $S_{Kn1}[0|R_U|C_1|M_{U0}]$, and M_D (the desired vector to be written to ChipU);

d. ChipS produces R_S, M_{QD} (processed by running M_D against M_{U0} using Q) and $S_{Kn2}[R_U|R_S|C_1|M_{QD}]$ if the inputs were valid, and 0 for all outputs if the inputs were not valid.

e. If values returned in d are non zero, then ChipU is considered authentic. System can then call ChipU's WriteA function with these values from d.

f. ChipU should return a 1 to indicate success. A 0 should only be returned if the data generated by ChipS is incorrect (e.g. a transmission error).

The choice of n1 and n2 must be such that ChipU's K_{n1} = ChipS's K_{n2}.

The data flow for this variant of authenticated writes is shown in Figure 331.

Note that Q in ChipS is part of ChipS's M. This allows a user to set up ChipS with a permission set for upgrades. This should be done to ChipS and that part of M designated by $P_{0..T-1}$ set to ReadOnly *before* ChipS is programmed with K_U. If K_S is programmed with K_U first, there is a risk of someone obtaining a half-setup ChipS and changing all of M_U instead of only the sections specified by Q.

In addition, CountRemaining in ChipS needs to be setup (including making it ReadOnly in P_S) before ChipS is programmed with K_U. ChipS should therefore be programmed to only perform a limited number of SignM operations (thereby limiting compromise exposure if a ChipS is stolen). Thus ChipS would itself need to be upgraded with a new CountRemaining every so often.

5.4.2.2 Indirect authenticated writes

This section describes an alternative authenticated write protocol when ChipU is more readily available to an attacker and ChipS is less available to an attacker. We can store different keys on ChipU and ChipS, and implement a mapping between them in such a way that if the attacker is able to obtain a key from a given ChipU, they cannot upgrade all ChipUs.

In the general case, this is accomplished by storing key K_S on ChipS, and K_U and f on ChipU. The relationship is $f(K_S) = K_U$ such that knowledge of K_U and f does not make it easy to determine K_S. This implies that a one-way function is desirable for f.

In the QA Chip domain, we define f as a number (e.g. 32-bits) such that $SHA1(K_S \mid f) = K_U$. The value of f (random between chips) can be stored in a known location within M_1 as a constant for the life of the QA Chip. It is possible to use the same f for multiple relationships if desired, since f is public and the protection lies in the fact that f varies between QA Chips (preferably in a non-predictable way).

The indirect protocol is the same as the direct protocol with the exception that f is additionally passed in to the SignM function so that ChipS is able to generate the correct key. The System obtains f by performing a Read of M_1. Note that all other functions, including the WriteA function in ChipU, are identical to their direct authentication counterparts.

614

SignM[f,n,V,W,X,Y,Z]   Advances R, and returns R, $S_{f(Kn)}[W|R|C_1|Z]$ only if $Y = S_{f(Kn)}[V|W|C_1|X]$ and CountRemaining > 0. Otherwise returns all 0s. The time taken to calculate and compare signatures must be independent of data content.

5   Before reading ChipU's memory M_0 (the pre-upgrade value), the System must extract f from ChipU by performing the following tasks:

a. System calls ChipU's Read function, passing in (dontCare, 1, dontCare)

b. ChipU returns M_1, from which System can extract f_U

c. System stores f_U for future use

10

To update ChipU's M vector, the protocol is identical to that described in the basic authenticated write protocol with the exception of steps c and d:

c. System calls ChipS's SignM function, passing in f_U, n2 (the key to be used in ChipS), 0 (as used in a), R_U, M_{U0}, $S_{Kn1}[0|R_U|C_1|M_{U0}]$, and M_D (the desired vector to be written to ChipU);

15   d. ChipS produces R_S and $S_{fU(Kn2)}[R_U|R_S|C_1|M_D]$ if the inputs were valid, and 0 for all outputs if the inputs were not valid.

In addition, the choice of n1 and n2 must be such that ChipU's K_{n1} = ChipS's $f_U(K_{n2})$.

20   Note that f_U is obtained from M_1 *without validation*. This is because there is nothing to be gained by subverting the value of f_U, (because then the signatures won't match).

From the System's perspective, the protocol would take on a form like the following pseudocode:

```
           dontCare, M_R, dontCare  ← ChipR.Read(dontCare,1, dontCare)
25         f_R = extract from M_R
           . . .
           R_U, M_U, SIG_U ← ChipU.Read(keyNumOnChipU,0, 0)
           R_S, SIG_S = ChipS.SignM2(f_R, keyNumOnChipS, 0, R_U, M_U, SIG_U, M_D)
           If (R_S = SIG_S = 0)
30             // ChipU and therefore M_U is not to be trusted
           Else
               // ChipU and therefore M_U can be trusted
               ok = ChipU.WriteA(keyNumOnChipU, R_S, M_D, SIG_S)
               If (ok)
35                 // updating of data in ChipU was successful
               Else
                   // transmission error during WriteA
```

615

```
        EndIf
        EndIf
```

5.4.2.2.1 variant example

5   The indirect form of the example from Section 5.4.2.1.1 is shown here.

SignM[f,n,V,W,X,Y,Z] Advances R, decrements CountRemaining and returns R, Z_{QX} (Z applied to X
        with permissions Q), $S_{f(Kn)}[W|R|C_1|Z_{QX}]$ only if $Y = S_{f(Kn)}[V|W|C_1|X]$ and
        CountRemaining > 0. Otherwise returns all 0s. The time taken to calculate and
        compare signatures must be independent of data content.

10

Before reading ChipU's memory M_0 (the pre-upgrade value), the System must extract f from ChipU
by performing the following tasks:

a. System calls ChipU's Read function, passing in (dontCare, 1, dontCare)

b. ChipU returns M_1, from which System can extract f_U

15   c. System stores f_U for future use

To update ChipU's M vector, the protocol is identical to that described in the basic authenticated
write protocol with the exception of steps c and d:

c. System calls ChipS's SignM function, passing in f_U, n2 (the key to be used in ChipS), 0 (as used

20   in a), R_U, M_{U0}, $S_{Kn1}[0|R_U|C_1|M_{U0}]$, and M_D (the desired vector to be written to ChipU);

d. ChipS produces R_S, M_{QD} (processed by running M_D against M_{U0} using Q) and
    $S_{f_U(Kn2)}[R_U|R_S|C_1|M_{QD}]$ if the inputs were valid, and 0 for all outputs if the inputs were not valid.

In addition, the choice of n1 and n2 must be such that ChipU's $K_{n1} = $ ChipS's $f_U(K_{n2})$.

25

Note that f_U is obtained from M_1 *without validation*. This is because there is nothing to be gained by
subverting the value of f_U, (because then the signatures won't match).

From the System's perspective, the protocol would take on a form like the following pseudocode:

```
30              dontCare, M_R, dontCare ← ChipR.Read(dontCare,1, dontCare)
                f_R = extract from M_R
                . . .
                R_U, M_U, SIG_U ← ChipU.Read(keyNumOnChipU,0, 0)
                R_S, M_QD, SIG_S = ChipS.SignM2(f_R, keyNumOnChipS, 0, R_U, M_U, SIG_U, M_D)
35              If (R_S = M_QD = SIG_S = 0)
                    // ChipU and therefore M_U is not to be trusted
                Else
```

```
        // ChipU and therefore M_U can be trusted
        ok = ChipU.WriteA(keyNumOnChipU, R_S, M_QD, SIG_S)
        If (ok)
            // updating of data in ChipU was successful
        Else
            // transmission error during WriteA
        EndIf
    EndIf
```

5.4.3   Updating permissions for future writes

In order to reduce exposure to accidental and malicious attacks on P (and certain parts of M), only authorized users are allowed to update P. Writes to P are the same as authorized writes to M, except that they update P_n instead of M. Initially (at manufacture), P is set to be Read/Write for all M. As different processes fill up different parts of M, they can be sealed against future change by updating the permissions. Updating a chip's $P_{0..T-1}$ changes permissions for unauthorized writes to M_n, and updating $P_{T..T+N-1}$ changes permissions for authorized writes with key K_n.

P_n is only allowed to change to be a more restrictive form of itself. For example, initially all parts of M have permissions of Read/Write. A permission of Read/Write can be updated to Decrement Only or Read Only. A permission of Decrement Only can be updated to become Read Only. A Read Only permission cannot be further restricted.

In this transaction protocol, the System's chip is referred to as ChipS, and the chip being updated is referred to as ChipU. Each chip distrusts the other.

The protocol requires the following publicly available functions in ChipU:

Random[]   Returns R (does not advance R).

SetPermission[n,p,X,Y,Z]   Advances R, and updates P_p according to Y and returns 1 followed by the resultant P_p only if $S_{Kn}[R|X|Y|C_2] = Z$. Otherwise returns 0. P_p can only become more restricted. Passing in 0 for any permission leaves it unchanged (passing in Y=0 returns the current P_p).

Authenticated writes of permissions require that the System has access to a ChipS that is capable of generating appropriate signatures. ChipS requires the following variable:

CountRemaining   Part of ChipS's M_0 that contains the number of signatures that ChipS is allowed to generate. Decrements with each successful call to SignM and SignP. Permissions in ChipS's $P_{0..T-1}$ for this part of M_0 needs to be ReadOnly

once ChipS has been setup. Therefore CountRemaining can only be updated by another ChipS that will perform updates to that part of M_0 (assuming ChipS's P_n allows that part of M_0 to be updated).

5   In addition, ChipS requires either of the following two SignP functions depending on whether direct or indirect key storage is used (see direct vs indirect authenticated write protocols in Section 5.4.2):

SignP[n,X,Y]   Used when the same key is directly stored in both ChipS and ChipU. Advances R, decrements CountRemaining and returns R and $S_{Kn}[X|R|Y|C_2]$ only if CountRemaining > 0. Otherwise returns all 0s. The time taken to calculate and

10   compare signatures must be independent of data content.

SignP[f,n,X,Y]   Used when the same key is not directly stored in both ChipS and ChipU. In this case ChipU's K_{n1} = ChipS's $f(K_{n2})$. The function is identical to the direct form of SignP, except that it additionally accepts f and returns $S_{f(Kn)}[X|R|Y|C_2]$ instead of $S_{Kn}[X|R|Y|C_2]$.

15

5.4.3.1   Direct form of SignP

When the direct form of SignP is used, ChipU's P_n is updated as follows:

a.   System calls ChipU's Random function;

b.   ChipU returns R_U to System;

20   c.   System calls ChipS's SignP function, passing in n2, R_U and P_D (the desired P to be written to ChipU);

d.   ChipS produces R_S and $S_{Kn2}[R_U|R_S|P_D|C_2]$ if it is still permitted to produce signatures.

e.   If values returned in d are non zero, then System can then call ChipU's SetPermission function with n1, the desired permission entry p, R_S, P_D and $S_{Kn2}[R_U|R_S|P_D|C_2]$.

25   f.   ChipU verifies the received signature against its own generated signature $S_{Kn1}[R_U|R_S|P_D|C_2]$ and applies P_D to P_n if the signature matches

g.   System checks 1st output parameter. 1 = success, 0 = failure.

The choice of n1 and n2 must be such that ChipU's K_{n1} = ChipS's K_{n2}.

30

The data flow for basic authenticated writes to permissions is shown in Figure 332.

5.4.3.2   Indirect form of SignP

When the indirect form of SignP is used in ChipS, the System must extract f from ChipU (so it

35   knows how to generate the correct key) by performing the following tasks:

a.   System calls ChipU's Read function, passing in (dontCare, 1, dontCare)

b.   ChipU returns M_1, from which System can extract f_U

c. System stores f_U for future use

ChipU's P_n is updated as follows:

a. System calls ChipU's Random function;

b. ChipU returns R_U to System;

c. System calls ChipS's SignP function, passing in f_U, n2, R_U and P_D (the desired P to be written to ChipU);

d. ChipS produces R_S and $S_{fU(Kn2)}[R_U|R_S|P_D|C_2]$ if it is still permitted to produce signatures.

e. If values returned in d are non zero, then System can then call ChipU's SetPermission function with n1, the desired permission entry p, R_S, P_D and $S_{fU(Kn2)}[R_U|R_S|P_D|C_2]$.

f. ChipU verifies the received signature against $S_{Kn1}[R_U|R_S|P_D|C_2]$ and applies P_D to P_n if the signature matches

g. System checks 1st output parameter. 1 = success, 0 = failure.

In addition, the choice of n1 and n2 must be such that ChipU's K_{n1} = ChipS's $f_U(K_{n2})$.

5.4.4   Protecting memory vectors

To protect the appropriate part of M_n against unauthorized writes, call SetPermissions[n] for n = 0 to T-1. To protect the appropriate part of M_0 against authorized writes with key n, call SetPermissions[T+n] for n=0 to N-1.

Note that only M_0 can be written in an authenticated fashion.

Note that the SetPermission function must be called *after* the part of M has been set to the desired value.

For example, if adding a serial number to an area of M_1 that is currently ReadWrite so that noone is permitted to update the number again:

- the Write function is called to write the serial number to M_1

- SetPermission(1) is called for to set that part of M to be ReadOnly for non-authorized writes.

If adding a consumable value to M_0 such that only keys 1-2 can update it, and keys 0, and 3-N cannot:

- the Write function is called to write the amount of consumable to M

- SetPermission is called for 0 to set that part of M_0 to be DecrementOnly for *non-authorized* writes. This allows the amount of consumable to decrement.

- SetPermission is called for n = {T, T+3, T+4 ..., T+N-1} to set that part of M_0 to be ReadOnly for *authorized* writes using all but keys 1 and 2. This leaves keys 1 and 2 with ReadWrite permissions to M_0.

It is possible for someone who knows a key to further restrict other keys, but it is not in anyone's interest to do so.

5.5   PROGRAMMING K

In this case, we have a factory chip (*ChipF*) connected to a System. The System wants to program the key in another chip (*ChipP*). System wants to avoid passing the new key to ChipP in the clear, and also wants to avoid the possibility of the key-upgrade message being replayed on another ChipP (even if the user doesn't know the key).

5

The protocol assumes that ChipF and ChipP already share (directly or indirectly) a secret key K_{old}. This key is used to ensure that only a chip that knows K_{old} can set K_{new}.

Although the example shows a ChipF that is only allowed to program a specific number of ChipPs,

10 the key-upgrade protocol can be easily altered (similar to the way the write protocols have variants) to provide other means of limiting the ability to update ChipPs.

The protocol requires the following publicly available functions in ChipP:

Random[]   Returns R (does not advance R).

15   ReplaceKey[n, X, Y, Z] Replaces K_n by $S_{Kn}[R|X|C_3] \oplus Y$, advances R, and returns 1 only if $S_{Kn}[X|Y|C_3] = Z$. Otherwise returns 0. The time taken to calculate signatures and compare values must be identical for all inputs.

And the following data and functions in ChipF:

20 CountRemaining   Part of M_0 with contains the number of signatures that ChipF is allowed to generate. Decrements with each successful call to GetProgramKey. Permissions in P for this part of M_0 needs to be ReadOnly once ChipF has been setup. Therefore can only be updated by a ChipS that has authority to perform updates to that part of M_0.

25 K_{new} The new key to be transferred from ChipF to ChipP. Must not be visible. After manufacture, K_{new} is 0.

SetPartialKey[X]   Updates K_{new} to be $K_{new} \oplus X$. This function allows K_{new} to be programmed in any number of steps, thereby allowing different people or systems to know different parts of the key (but

30 not the whole K_{new}). K_{new} is stored in ChipF's flash memory.

In addition, ChipF requires either of the following GetProgramKey functions depending on whether direct or indirect key storage is used on the input key and/or output key (see direct vs indirect authenticated write protocols in Section 5.4.2):

35 GetProgramKey1[n, X]   Direct to direct. Used when the same key (K_n) is directly stored in both ChipF and ChipP and we want to store K_{new} in ChipP. Advances R_F, decrements CountRemaining, outputs R_F, the encrypted key $S_{Kn}[X|R_F|C_3] \oplus K_{new}$ and a

signature of the first two outputs plus C_3 if CountRemaining>0. Otherwise outputs 0. The time to calculate the encrypted key & signature must be identical for all inputs.

GetProgramKey2[f, n, X]   Direct to indirect. Used when the same key (K_n) is directly stored in both ChipF and ChipP but we want to store $f_P(K_{new})$ in ChipP instead of simply K_{new} (i.e. we want to keep the key in ChipP to be different in all ChipPs). In this case ChipP's K_{n1} = ChipF's $f_P(K_{n2})$. The function is identical to GetProgramKey1, except that it additionally accepts f_P, and returns $S_{Kn}[X|R_F|C_3]\oplus f_P(K_{new})$ instead of $S_{Kn}[X|R_F|C_3] \oplus K_{new}$. Note that the produced signature is produced using K_n since that is what is already stored in ChipP.

GetProgramKey3[f, n, X] Indirect to direct. Used when the same key is not directly stored in both ChipF and ChipP but we want to store K_{new} in ChipP. In this case ChipP's K_{n1} = ChipF's $f_P(K_{n2})$. The function is identical to GetProgramKey1, except that it additionally accepts f_P, and returns $S_{fP(Kn)}[X|R_F|C_3]\oplus K_{new}$ instead of $S_{Kn}[X|R_F|C_3]\oplus K_{new}$. The produced signature is produced using $f_P(Kn)$ instead of K_n since that is what is already stored in ChipP.

GetProgramKey4[f, n, X] Indirect to indirect. Used when the same key is not directly stored in both ChipF and ChipP but we want to store $f_P(K_{new})$ in ChipP instead of simply K_{new} (i.e. we want to keep the key in ChipP to be different in all ChipPs). In this case ChipP's K_{n1} = ChipF's $f_P(K_{n2})$. The function is identical to GetProgramKey3, except that it returns $S_{fP(Kn)}[X|R_F|C_3]\oplus f_P(K_{new})$ instead of $S_{fP(Kn)}[X|R_F|C_3]\oplus K_{new}$. The produced signature is produced using $f_P(K_n)$ since that is what is already stored in ChipP.

Since there are likely to be few ChipFs, and many ChipPs, the indirect forms of GetProgramKey can be usefully employed.

5.5.1   GetProgramKey1 - direct to direct

With the "old key = direct, new key = direct" form of GetProgramKey, to update P's key :

a.   System calls ChipP's Random function;

b.   ChipP returns R_P to System;

c.   System calls ChipF's GetProgramKey function, passing in n2 (the desired key to use) and the result from b;

d.   ChipF updates R_F, then calculates and returns R_F, $S_{Kn2}[R_P|R_F|C_3]\oplus K_{new}$, and $S_{Kn2}[R_F|S_{Kn2}[R_P|R_F|C_3]\oplus K_{new}|C_3]$;

e.   If the response from d is not 0, System calls ChipP's ReplaceKey function, passing in n1 (the key to use in ChipP) and the response from d;

f.   System checks response from ChipP. If the response is 1, then ChipP's K_{n1} has been correctly updated to K_{new}. If the response is 0, ChipP's K_{n1} has not been updated.

The choice of n1 and n2 must be such that ChipP's K_{n1} = ChipF's K_{n2}.

The data flow for key updates is shown in Figure 333:

Note that K_{new} is never passed in the open. An attacker could send its own R_P, but cannot produce $S_{Kn2}[R_P|R_F|C_3]$ without K_{n2}. The signature based on K_{new} is sent to ensure that ChipP will be able to determine if either of the first two parameters have been changed en route.

CountRemaining needs to be setup in M_{F0} (including making it ReadOnly in P) before ChipF is programmed with K_P. ChipF should therefore be programmed to only perform a limited number of GetProgramKey operations (thereby limiting compromise exposure if a ChipF is stolen). An authorized ChipS can be used to update this counter if necessary (see Section 5.4.2 on page 610).

5.5.2   GetProgramKey2 - direct to indirect

With the "old key = direct, new key = indirect" form of GetProgramKey, to update P's key, the System must extract f from ChipP (so it can tell ChipF how to generate the correct key) by performing the following tasks:

a. System calls ChipP's Read function, passing in (dontCare, 1, dontCare)

b. ChipP returns M_1, from which System can extract f_P

c. System stores f_P for future use

ChipP's key is updated as follows:

a. System calls ChipP's Random function;

b. ChipP returns R_P to System;

c. System calls ChipF's GetProgramKey function, passing in f_P, n2 (the desired key to use) and the result from b;

d. ChipF updates R_F, then calculates and returns R_F, $S_{Kn2}[R_P|R_F|C_3]\oplus f_P(K_{new})$, and $S_{Kn2}[R_F|S_{Kn2}[R_P|R_F|C_3]\oplus f_P(K_{new})|C_3]$;

e. If the response from d is not 0, System calls ChipP's ReplaceKey function, passing in n1 (the key to use in ChipP) and the response from d;

f. System checks response from ChipP. If the response is 1, then ChipP's K_{n1} has been correctly updated to $f_P(K_{new})$. If the response is 0, ChipP's K_{n1} has not been updated.

The choice of n1 and n2 must be such that ChipP's K_{n1} = ChipF's K_{n2}.

5.5.3 GetProgramKey3 - indirect to direct

With the "old key = indirect, new key = direct" form of GetProgramKey, to update P's key, the System must extract f from ChipP (so it can tell ChipF how to generate the correct key) by performing the following tasks:

a. System calls ChipP's Read function, passing in (dontCare, 1, dontCare)

b. ChipP returns M_1, from which System can extract f_P

c. System stores f_P for future use

ChipP's key is updated as follows:

a. System calls ChipP's Random function;

b. ChipP returns R_P to System;

c. System calls ChipF's GetProgramKey function, passing in f_P, n2 (the desired key to use) and the result from b;

d. ChipF updates R_F, then calculates and returns R_F, $S_{fP(Kn2)}[R_P|R_F|C_3] \oplus K_{new}$, and $S_{fP(Kn2)}[R_F|S_{fP(Kn2)}[R_P|R_F|C_3] \oplus K_{new}|C_3]$;

e. If the response from d is not 0, System calls ChipP's ReplaceKey function, passing in n1 (the key to use in ChipP) and the response from d;

f. System checks response from ChipP. If the response is 1, then ChipP's K_{n1} has been correctly updated to K_{new}. If the response is 0, ChipP's K_{n1} has not been updated.

The choice of n1 and n2 must be such that ChipP's K_{n1} = ChipF's $f_P(K_{n2})$.

5.5.4 GetProgramKey4 - indirect to indirect

With the "old key = indirect, new key = indirect" form of GetProgramKey, to update P's key, the System must extract f from ChipP (so it can tell ChipF how to generate the correct key) by performing the following tasks:

a. System calls ChipP's Read function, passing in (dontCare, 1, dontCare)

b. ChipP returns M_1, from which System can extract f_P

c. System stores f_P for future use

ChipP's key is updated as follows:

a. System calls ChipP's Random function;

b. ChipP returns R_P to System;

c. System calls ChipF's GetProgramKey function, passing in f_P, n2 (the desired key to use) and the result from b;

d. ChipF updates R_F, then calculates and returns R_F, $S_{fP(Kn2)}[R_P|R_F|C_3] \oplus f_P(K_{new})$, and $S_{fP(Kn2)}[R_F|S_{fP(Kn2)}[R_P|R_F|C_3] \oplus f_P(K_{new})|C_3]$;

623

e. If the response from d is not 0, System calls ChipP's ReplaceKey function, passing in n1 (the key to use in ChipP) and the response from d;

f. System checks response from ChipP. If the response is 1, then ChipP's K_{n1} has been correctly updated to $f_P(K_{new})$. If the response is 0, ChipP's K_{n1} has not been updated.

5   The choice of n1 and n2 must be such that ChipP's K_{n1} = ChipF's $f_P(K_{n2})$.

5.5.5   Chicken and Egg

The Program Key protocol requires both ChipF and ChipP to know K_{old} (either directly or indirectly). Obviously both chips had to be programmed in some way with K_{old}, and thus K_{old} can be thought of

10   as an older K_{new}: K_{old} can be placed in chips if another ChipF knows K_{older}, and so on.

Although this process allows a chain of reprogramming of keys, with each stage secure, at some stage the very first key (K_{first}) must be placed in the chips. K_{first} is in fact programmed with the chip's microcode at the manufacturing test station as the last step in manufacturing test. K_{first} can be a

15   manufacturing batch key, changed for each batch or for each customer etc., and can have as short a life as desired. Compromising K_{first} need not result in a complete compromise of the chain of Ks. This is especially true if K_{first} is indirectly stored in ChipPs (i.e. each ChipP holds an f and $f(K_{first})$ instead of K_{first} directly). One example is where K_{first} (the key stored in each chip after manufacture/test) is a batch key, and can be different per chip. K_{first} may advance to a ComCo

20   specific K_{second} etc. but still remain indirect. A direct form (e.g. K_{final}) only needs to go in if it is actually required at the end of the programming chain.

Depending on reprogramming requirements, K_{first} can be the same or different for all K_n.

25   ## 6   Memjet forms of Protocols

Physical QA Chips are used in Memjet printer systems to store printer operating parameters as well as consumable parameters.

6.1   PRINTER_QA

30   A PRINTER_QA is stored within each print engine to perform two primary tasks:

- storage and protection of operating parameters
- a means of indirect read validation of other QA Chip data vectors

Each PRINTER_QA contains the following keys:

Table 229. Keys in PrinterQA

Key	Contents	Comments
0	Upgrade Key	Used to upgrade the operating parameters. Should be indirect form of key (i.e. a different key for each PRINTER_QA) so that an indirect form of the write is required.
1	Consumable Read Validation Key	Used to indirectly read the data from an CONSUMABLE_QA chip using indirect authenticated read protocol (Section 5.3.2 on page 606).
2	PrintEngineController Read Validation Key	When reading data from the PRINTER_QA, the system can either trust the data, or must use this key to perform the authenticated read protocol (see Section 5.3 on page 604).
3-n	(reserved)	Currently unused. Could be used to provide a means to indirectly read additional print engine operating parameters ala K1, or provide additional Print Engine validation ala K2.

5

Note that if multiple Print Engine Controllers are used (e.g. a multiple SoPEC system), then multiple PrintEngineController Read Validation Keys are required. These keys can be stored within a single PRINTER_QA (e.g. in K_3 and beyond), or can be stored in separate PRINTER_QAs (for example each SoPEC (or group of SoPECs) has an individual PRINTER_QA).

10

The functions required in the PRINTER_QA are:

• Random, ReplaceKey, to allow key programming & substitution

• Read, to allow reads of data

• Write, to allow updates of M_{1+} during manufacture

15 • WriteAuth, to provide a means of updating the M_0 data (operating parameters)

- SetPermissions, to provide a means of updating write permissions
- Test, to provide a means of checking if consumable reads are valid
- Translate, to provide a means of indirect reading of consumable data

5 6.2   CONSUMABLE_QA

A CONSUMABLE_QA is stored with each consumable (e.g. ink cartridge) to perform two primary tasks:

- storage of consumable related data
- protection of consumable amount remaining

10

Each CONSUMABLE_QA contains the following keys:

Table 230. Keys in CONSUMABLE_QA

Key	Contents	Comments
0	Upgrade Key	Used to upgrade the consumable parameters. Should be stored as the indirect form of the key (i.e. a different key for each CONSUMABLE_QA) so that an indirect form of the write is required.
1	Consumable Read Validation Key	When reading data from the CONSUMABLE_QA, the system can either trust the data, or must use this key to perform either the direct or indirect authenticated read protocol (see Section 5.3 on page 604).
2	(reserved)	Currently unused.
3-n	(reserved)	Currently unused.

15 The functions required in the CONSUMABLE_QA are:

- Random, ReplaceKey, to allow key programming & substitution
- Read, to allow reads of data
- Write, to allow updates of M_{1+} during manufacture
- WriteAuth, to provide a means of updating the M_0 data (consumable remaining)

20 - SetPermissions, to provide a means of updating write permissions

626

AUTHENTICATION OF CONSUMABLES

1   Introduction

Manufacturers of systems that require consumables (such as a laser printer that requires toner
cartridges) have struggled with the problem of authenticating consumables, to varying levels of
success. Most have resorted to specialized packaging that involves a patent. However this does not
stop home refill operations or clone manufacture in countries with weak industrial property
protection. The prevention of copying is important to prevent poorly manufactured substitute
consumables from damaging the base system. For example, poorly filtered ink may clog print
nozzles in an ink jet printer, causing the consumer to blame the system manufacturer and not admit
the use of non-authorized consumables.

To solve the authentication problem, this document describes an QA Chip that contains
authentication keys and circuitry specially designed to prevent copying. The chip is manufactured
using the standard Flash memory manufacturing process, and is low cost enough to be included in
consumables such as ink and toner cartridges. The implementation is approximately $1mm^2$ in a
0.25 micron flash process, and has an expected manufacturing cost of approximately 10 cents in
2003.

2   NSA

Once programmed, the QA Chips as described here are compliant with the NSA export guidelines
since they do not constitute a strong encryption device. They can therefore be practically
manufactured in the USA (and exported) or anywhere else in the world.

3   Nomenclature

The following symbolic nomenclature is used throughout this document:

Table 231. Summary of symbolic nomenclature

Symbol	Description
F[X]	Function F, taking a single parameter X
F[X, Y]	Function F, taking two parameters, X and Y
X | Y	X concatenated with Y
X ∧ Y	Bitwise X AND Y
X ∨ Y	Bitwise X OR Y (inclusive-OR)
X ⊕ Y	Bitwise X XOR Y (exclusive-OR)
¬X	Bitwise NOT X (complement)

X ← Y	X is assigned the value Y
X ← {Y, Z}	The domain of assignment inputs to X is Y and Z
X = Y	X is equal to Y
X ≠ Y	X is not equal to Y
⇓X	Decrement X by 1 (floor 0)
⇑X	Increment X by 1 (modulo register length)
Erase X	Erase Flash memory register X
SetBits[X, Y]	Set the bits of the Flash memory register X based on Y
Z ← ShiftRight[X, Y]	Shift register X right one bit position, taking input bit from Y and placing the output bit in Z

4 PSEUDOCODE

4.1.1 Asynchronous

The following pseudocode:

```
var = expression
```

means the var signal or output is equal to the evaluation of the expression.

4.1.2 Synchronous

The following pseudocode:

```
var ← expression
```

means the var register is assigned the result of evaluating the expression during this cycle.

4.1.3 Expression

Expressions are defined using the nomenclature in Table 231 above. Therefore:

```
var = (a = b)
```

is interpreted as the var signal is 1 if a is equal to b, and 0 otherwise.

4.2 DIAGRAMS

Black is used to denote data, and red to denote 1-bit control-signal lines.

4.3 QA CHIP TERMINOLOGY

This document refers to QA Chips by their function in particular protocols:

- For authenticated reads, ChipA is the QA Chip being authenticated, and ChipT is the QA Chip that is trusted.
- For replacement of keys, ChipP is the QA Chip being programmed with the new key, and ChipF is the factory QA Chip that generates the message to program the new key.
- For upgrades of data in a QA Chip, ChipU is the QA Chip being upgraded, and ChipS is the QA Chip that signs the upgrade value.

Any given physical QA Chip will contain functionality that allows it to operate as an entity in some number of these protocols.

628

Therefore, wherever the terms ChipA, ChipT, ChipP, ChipF, ChipU and ChipS are used in this document, they are referring to *logical* entities involved in an authentication protocol as defined in subsequent sections.

5   *Physical* QA Chips are referred to by their location. For example, each ink cartridge may contain a QA Chip referred to as an INK_QA, with all INK_QA chips being on the same physical bus. In the same way, the QA Chip inside a printer is referred to as PRINTER_QA, and will be on a separate bus to the INK_QA chips.

10   5   Concepts and Terms

This chapter provides a background to the problem of authenticating consumables. For more in-depth introductory texts, see [12], [78], and [56].

5.1   BASIC TERMS

15   A message, denoted by M, is *plaintext*. The process of transforming M into *ciphertext* C, where the substance of M is hidden, is called *encryption*. The process of transforming C back into M is called *decryption*. Referring to the encryption function as E, and the decryption function as D, we have the following identities:

$$E[M] = C$$
$$D[C] = M$$

20

Therefore the following identity is true:

$$D[E[M]] = M$$

5.2   SYMMETRIC CRYPTOGRAPHY

A symmetric encryption algorithm is one where:

25   •   the encryption function E relies on key K_1,

•   the decryption function D relies on key K_2,

• K_2 can be derived from K_1, and

• K_1 can be derived from K_2.

In most symmetric algorithms, K_1 equals K_2. However, even if K_1 does not equal K_2, given that one

30   key can be derived from the other, a single key K can suffice for the mathematical definition. Thus:

$$E_K[M] = C$$
$$D_K[C] = M$$

629

The security of these algorithms rests very much in the key K. Knowledge of K allows *anyone* to encrypt or decrypt. Consequently K must remain a secret for the duration of the value of M. For example, M may be a wartime message "My current position is grid position 123-456". Once the war is over the value of M is greatly reduced, and if K is made public, the knowledge of the combat

5   unit's position may be of no relevance whatsoever. Of course if it is politically sensitive for the combat unit's position to be known even after the war, K may have to remain secret for a very long time.

An enormous variety of symmetric algorithms exist, from the textbooks of ancient history through to

10   sophisticated modern algorithms. Many of these are insecure, in that modern cryptanalysis techniques (see Section 5.7 on page 646) can successfully attack the algorithm to the extent that K can be derived.

The security of the particular symmetric algorithm is a function of two things: the strength of the

15   algorithm and the length of the key [78].

The strength of an algorithm is difficult to quantify, relying on its resistance to cryptographic attacks (see Section 5.7 on page 646). In addition, the longer that an algorithm has remained in the public eye, and yet remained unbroken in the midst of intense scrutiny, the more secure the algorithm is

20   likely to be. By contrast, a secret algorithm that has not been scrutinized by cryptographic experts is unlikely to be secure.

Even if the algorithm is "perfectly" strong (the only way to break it is to try every key - see Section 5.7.1.5 on page 647), eventually the right key will be found. However, the more keys there are, the

25   more keys have to be tried. If there are N keys, it will take a maximum of N tries. If the key is N bits long, it will take a maximum of 2^N tries, with a 50% chance of finding the key after only half the attempts (2^{N-1}). The longer N becomes, the longer it will take to find the key, and hence the more secure it is. What makes a good key length depends on the value of the secret and the time for which the secret must remain secret as well as available computing resources.

30

In 1996, an ad hoc group of world-renowned cryptographers and computer scientists released a report [9] describing minimal key lengths for symmetric ciphers to provide adequate commercial security. They suggest an absolute minimum key length of 90 bits in order to protect data for 20 years, and stress that increasingly, as cryptosystems succumb to smarter attacks than brute-force

35   key search, even more bits may be required to account for future surprises in cryptanalysis techniques.

We will ignore most historical symmetric algorithms on the grounds that they are insecure, especially given modern computing technology. Instead, we will discuss the following algorithms:

- DES
- Blowfish
- RC5
- IDEA

5.2.1   DES

DES (Data Encryption Standard) [26] is a US and international standard, where the same key is used to encrypt and decrypt. The key length is 56 bits. It has been implemented in hardware and software, although the original design was for hardware only. The original algorithm used in DES was patented in 1976 (US patent number 3,962,539) and has since expired.

During the design of DES, the NSA (National Security Agency) provided secret S-boxes to perform the key-dependent nonlinear transformations of the data block. After differential cryptanalysis was discovered outside the NSA, it was revealed that the DES S-boxes were specifically designed to be resistant to differential cryptanalysis.

As described in [95], using 1993 technology, a 56-bit DES key can be recovered by a custom-designed $1 million machine performing a brute force attack in only 35 minutes. For $10 million, the key can be recovered in only 3.5 minutes. DES is clearly not secure now, and will become less so in the future.

A variant of DES, called *triple-DES* is more secure, but requires 3 keys: K_1, K_2, and K_3. The keys are used in the following manner:

$$E_{K3}[D_{K2}[E_{K1}[M]]] = C$$
$$D_{K3}[E_{K2}[D_{K1}[C]]] = M$$

The main advantage of triple-DES is that existing DES implementations can be used to give more security than single key DES. Specifically, triple-DES gives protection of equivalent key length of 112 bits [78]. Triple-DES does not give the equivalent protection of a 168-bit key (3 × 56) as one might naively expect.

Equipment that performs triple-DES decoding and/or encoding cannot be exported from the United States.

5.2.2   Blowfish

Blowfish is a symmetric block cipher first presented by Schneier in 1994 [76]. It takes a variable length key, from 32 bits to 448 bits, is unpatented, and is both license and royalty free. In addition, it is much faster than DES.

The Blowfish algorithm consists of two parts: a key-expansion part and a data-encryption part. Key expansion converts a key of at most 448 bits into several subkey arrays totaling 4168 bytes. Data

encryption occurs via a 16-round Feistel network. All operations are XORs and additions on 32-bit words, with four index array lookups per round.

It should be noted that decryption is the same as encryption except that the subkey arrays are used in the reverse order. Complexity of implementation is therefore reduced compared to other algorithms that do not have such symmetry.

[77] describes the published attacks which have been mounted on Blowfish, although the algorithm remains secure as of February 1998 [79]. The major finding with these attacks has been the discovery of certain weak keys. These weak keys can be tested for during key generation. For more information, refer to [77] and [79].

5.2.3   RC5

Designed by Ron Rivest in 1995, RC5 [74] has a variable block size, key size, and number of rounds. Typically, however, it uses a 64-bit block size and a 128-bit key.

The RC5 algorithm consists of two parts: a key-expansion part and a data-encryption part. Key expansion converts a key into $2r+2$ subkeys (where r = the number of rounds), each subkey being w bits. For a 64-bit blocksize with 16 rounds ($w=32$, $r=16$), the subkey arrays total 136 bytes. Data encryption uses addition mod 2^w, XOR and bitwise rotation.

An initial examination by Kaliski and Yin [43] suggested that standard linear and differential cryptanalysis appeared impractical for the 64-bit blocksize version of the algorithm. Their differential attacks on 9 and 12 round RC5 require 2^{45} and 2^{62} chosen plaintexts respectively, while the linear attacks on 4, 5, and 6 round RC5 requires 2^{37}, 2^{47} and 2^{57} known plaintexts). These two attacks are independent of key size.

More recently however, Knudsen and Meier [47] described a new type of differential attack on RC5 that improved the earlier results by a factor of 128, showing that RC5 has certain weak keys.

RC5 is protected by multiple patents owned by RSA Laboratories. A license must be obtained to use it.

5.2.4   IDEA

Developed in 1990 by Lai and Massey [53], the first incarnation of the IDEA cipher was called PES. After differential cryptanalysis was discovered by Biham and Shamir in 1991, the algorithm was strengthened, with the result being published in 1992 as IDEA [52].

IDEA uses 128-bit keys to operate on 64-bit plaintext blocks. The same algorithm is used for encryption and decryption. It is generally regarded as the most secure block algorithm available today [78][78].

The biggest drawback of IDEA is the fact that it is patented (US patent number 5,214,703, issued in 1993), and a license must be obtained from Ascom Tech AG (Bern) to use it.

5.3   ASYMMETRIC CRYPTOGRAPHY

An asymmetric encryption algorithm is one where:

- the encryption function E relies on key K_1,
- the decryption function D relies on key K_2,
- K_2 cannot be derived from K_1 in a reasonable amount of time, and
- K_1 cannot be derived from K_2 in a reasonable amount of time.

5   Thus:

$$E_{K1}[M] = C$$
$$D_{K2}[C] = M$$

These algorithms are also called *public-key* because one key K_1 can be made public. Thus anyone can encrypt a message (using K_1) but only the person with the corresponding decryption key (K_2) can decrypt and thus read the message.

10   In most cases, the following identity also holds:

$$E_{K2}[M] = C$$
$$D_{K1}[C] = M$$

This identity is very important because it implies that anyone with the public key K_1 can see M and know that it came from the owner of K_2. No-one else could have generated C because to do so

15   would imply knowledge of K_2. This gives rise to a different application, unrelated to encryption - digital signatures.

The property of not being able to derive K_1 from K_2 and vice versa in a reasonable time is of course clouded by the concept of *reasonable time*. What has been demonstrated time after time, is that a

20   calculation that was thought to require a long time has been made possible by the introduction of faster computers, new algorithms etc. The security of asymmetric algorithms is based on the difficulty of one of two problems: factoring large numbers (more specifically large numbers that are the product of two large primes), and the difficulty of calculating discrete logarithms in a finite field. Factoring large numbers is conjectured to be a hard problem given today's understanding of

25   mathematics. The problem however, is that factoring is getting easier much faster than anticipated. Ron Rivest in 1977 said that factoring a 125-digit number would take 40 quadrillion years [30]. In 1994 a 129-digit number was factored [3]. According to Schneier, you need a 1024-bit number to get the level of security today that you got from a 512-bit number in the 1980s [78]. If the key is to last for some years then 1024 bits may not even be enough. Rivest revised his key length estimates

30   in 1990: he suggests 1628 bits for high security lasting until 2005, and 1884 bits for high security lasting until 2015 [69]. Schneier suggests 2048 bits are required in order to protect against corporations and governments until 2015 [80].

Public key cryptography was invented in 1976 by Diffie and Hellman [15][15], and independently by Merkle [57]. Although Diffie, Hellman and Merkle patented the concepts (US patent numbers 4,200,770 and 4,218,582), these patents expired in 1997.

A number of public key cryptographic algorithms exist. Most are impractical to implement, and many generate a very large C for a given M or require enormous keys. Still others, while secure, are far too slow to be practical for several years. Because of this, many public key systems are hybrid - a public key mechanism is used to transmit a symmetric session key, and then the session key is used for the actual messages.

All of the algorithms have a problem in terms of key selection. A random number is simply not secure enough. The two large primes p and q must be chosen carefully - there are certain weak combinations that can be factored more easily (some of the weak keys can be tested for). But nonetheless, key selection is not a simple matter of randomly selecting 1024 bits for example. Consequently the key selection process must also be secure.

Of the practical algorithms in use under public scrutiny, the following are discussed:

- RSA
- DSA
- ElGamal

5.3.1   RSA

The RSA cryptosystem [75], named after Rivest, Shamir, and Adleman, is the most widely used public key cryptosystem, and is a de facto standard in much of the world [78].

The security of RSA depends on the conjectured difficulty of factoring large numbers that are the product of two primes (p and q). There are a number of restrictions on the generation of p and q. They should both be large, with a similar number of bits, yet not be close to one another (otherwise $p \equiv q \equiv \sqrt{pq}$). In addition, many authors have suggested that p and q should be strong primes [56]. The Hellman-Bach patent (US patent number 4,633,036) covers a method for generating strong RSA primes p and q such that $n = pq$ and factoring n is believed to be computationally infeasible. The RSA algorithm patent was issued in 1983 (US patent number 4,405,829). The patent expires on September 20, 2000.

5.3.2   DSA

DSA (Digital Signature Algorithm) is an algorithm designed as part of the Digital Signature Standard (DSS) [29]. As defined, it cannot be used for generalized encryption. In addition, compared to RSA, DSA is 10 to 40 times slower for signature verification [40]. DSA explicitly uses the SHA-1 hashing algorithm (see Section 5.5.3.3 on page 640).

DSA key generation relies on finding two primes p and q such that q divides p-1. According to Schneier [78], a 1024-bit p value is required for long term DSA security. However the DSA standard [29] does not permit values of p larger than 1024 bits (p must also be a multiple of 64 bits).

The US Government owns the DSA algorithm and has at least one relevant patent (US patent 5,231,688 granted in 1993). However, according to NIST [61]:

> "The DSA patent and any foreign counterparts that may issue are available for use without any written permission from or any payment of royalties to the U.S. government."

In a much stronger declaration, NIST states in the same document [61] that DSA does not infringe third party's rights:

> "NIST reviewed all of the asserted patents and concluded that none of them would be infringed by DSS. Extra protection will be written into the PK1 pilot project that will prevent an organization or individual from suing anyone except the government for patent infringement during the course of the project."

It must however, be noted that the Schnorr authentication algorithm [81] (US patent 4,995,082) patent holder claims that DSA infringes his patent. The Schnorr patent is not due to expire until 2008.

5.3.3   ElGamal

The ElGamal scheme [22][22] is used for both encryption and digital signatures. The security is based on the conjectured difficulty of calculating discrete logarithms in a finite field.

Key selection involves the selection of a prime p, and two random numbers g and x such that both g and x are less than p. Then calculate $y = gx$ mod p. The public key is y, g, and p. The private key is x.

ElGamal is unpatented. Although it uses the patented Diffie-Hellman public key algorithm [15][15], those patents expired in 1997. ElGamal public key encryption and digital signatures can now be safely used without infringing third party patents.

5.4   CRYPTOGRAPHIC CHALLENGE-RESPONSE PROTOCOLS AND ZERO KNOWLEDGE PROOFS

The general principle of a challenge-response protocol is to provide identity authentication. The simplest form of challenge-response takes the form of a secret password. A asks B for the secret password, and if B responds with the correct password, A declares B authentic.

There are three main problems with this kind of simplistic protocol. Firstly, once B has responded with the password, any observer C will know what the password is. Secondly, A must know the password in order to verify it. Thirdly, if C impersonates A, then B will give the password to C (thinking C was A), thus compromising the password.

Using a copyright text (such as a *haiku*) as the password is not sufficient, because we are assuming that anyone is able to copy the password (for example in a country where intellectual property is not respected).

The idea of *cryptographic challenge-response protocols* is that one entity (the claimant) proves its identity to another (the verifier) by demonstrating knowledge of a secret known to be associated with that entity, *without revealing the secret itself* to the verifier during the protocol [56]. In the generalized case of cryptographic challenge-response protocols, with some schemes the verifier knows the secret, while in others the secret is not even known by the verifier. A good overview of these protocols can be found in [25], [78], and [56].

Since this documentation specifically concerns Authentication, the actual cryptographic challenge-response protocols used for authentication are detailed in the appropriate sections. However the concept of Zero Knowledge Proofs bears mentioning here.

The Zero Knowledge Proof protocol, first described by Feige, Fiat and Shamir in [24] is extensively used in Smart Cards for the purpose of authentication [34][34][34]. The protocol's effectiveness is based on the assumption that it is computationally infeasible to compute square roots modulo a large composite integer with unknown factorization. This is provably equivalent to the assumption that factoring large integers is difficult.

It should be noted that there is no need for the claimant to have significant computing power. Smart cards implement this kind of authentication using only a few modulo multiplications [34][34].

Finally, it should be noted that the Zero Knowledge Proof protocol is patented [82] (US patent 4,748,668, issued May 31, 1988).

5.5 ONE-WAY FUNCTIONS

A one-way function F operates on an input X, and returns F[X] such that X cannot be determined from F[X]. When there is no restriction on the format of X, and F[X] contains fewer bits than X, then collisions must exist. A collision is defined as two different X input values producing the same F[X] value - i.e. X_1 and X_2 exist such that $X_1 \neq X_2$ yet $F[X_1] = F[X_2]$.

When X contains more bits than F[X], the input must be compressed in some way to create the output. In many cases, X is broken into blocks of a particular size, and compressed over a number of rounds, with the output of one round being the input to the next. The output of the hash function is the last output once X has been consumed. A *pseudo-collision* of the compression function CF is defined as two different initial values V_1 and V_2 and two inputs X_1 and X_2 (possibly identical) are given such that $CF(V_1, X_1) = CF(V_2, X_2)$. Note that the existence of a pseudo-collision does not mean that it is easy to compute an X_2 for a given X_1.

636

We are only interested in one-way functions that are fast to compute. In addition, we are only interested in *deterministic* one-way functions that are repeatable in different implementations. Consider an example F where F[X] is the time between calls to F. For a given F[X] X cannot be determined because X is not even used by F. However the output from F will be different for different implementations. This kind of F is therefore not of interest.

In the scope of this document, we are interested in the following forms of one-way functions:
- Encryption using an unknown key
- Random number sequences
- Hash Functions
- Message Authentication Codes

5.5.1   Encryption using an unknown key

When a message is encrypted using an unknown key K, the encryption function E is effectively one-way. Without the key, it is computationally infeasible to obtain M from EK[M] without K. An encryption function is only one-way for as long as the key remains hidden.

An encryption algorithm does not create collisions, since E creates EK[M] such that it is possible to reconstruct M using function D. Consequently F[X] contains at least as many bits as X (no information is lost) if the one-way function F is E.

Symmetric encryption algorithms (see Section 5.2 on page 629) have the advantage over asymmetric algorithms (see Section 5.3 on page 632) for producing one-way functions based on encryption for the following reasons:
- The key for a given strength encryption algorithm is shorter for a symmetric algorithm than an asymmetric algorithm
- Symmetric algorithms are faster to compute and require less software or silicon

Note however, that the selection of a good key depends on the encryption algorithm chosen. Certain keys are not strong for particular encryption algorithms, so any key needs to be tested for strength. The more tests that need to be performed for key selection, the less likely the key will remain hidden.

5.5.2   Random number sequences

Consider a random number sequence R_0, R_1, ..., R_i, R_{i+1}. We define the one-way function F such that F[X] returns the Xth random number in the random sequence. However we must ensure that F[X] is repeatable for a given X on different implementations. The random number sequence

therefore cannot be truly random. Instead, it must be pseudo-random, with the generator making use of a specific seed.

There are a large number of issues concerned with defining good random number generators. Knuth, in [48] describes what makes a generator "good" (including statistical tests), and the general problems associated with constructing them. Moreau gives a high level survey of the current state of the field in [60].

The majority of random number generators produce the ith random number from the i-1th state - the only way to determine the ith number is to iterate from the 0th number to the ith. If i is large, it may not be practical to wait for i iterations.

However there is a type of random number generator that *does* allow random access. In [10], Blum, Blum and Shub define the ideal generator as follows: "... *we would like a pseudo-random sequence generator to quickly produce, from short seeds, long sequences (of bits) that appear in every way to be generated by successive flips of a fair coin*". They defined the x^2 *mod n* generator [10], more commonly referred to as the BBS generator. They showed that given certain assumptions upon which modern cryptography relies, a BBS generator passes extremely stringent statistical tests.

The BBS generator relies on selecting n which is a Blum integer ($n = pq$ where p and q are large prime numbers, $p \neq q$, p mod 4 = 3, and q mod 4 = 3). The initial state of the generator is given by x_0 where $x_0 = x^2$ mod n, and x is a random integer relatively prime to n. The ith pseudo-random bit is the least significant bit of x_i where:

$$x_i = x_{i-1}^2 \bmod n$$

As an extra property, knowledge of p and q allows a direct calculation of the ith number in the sequence as follows:

$$x_i = x_0^y \bmod n \qquad \text{where} \quad y = 2^i \bmod ((p-1)(q-1))$$

Without knowledge of p and q, the generator must iterate (the security of calculation relies on the conjectured difficulty of factoring large numbers).

When first defined, the primary problem with the BBS generator was the amount of work required for a single output bit. The algorithm was considered too slow for most applications. However the

advent of Montgomery reduction arithmetic [58] has given rise to more practical implementations, such as [59]. In addition, Vazirani and Vazirani have shown in [93] that depending on the size of n, more bits can safely be taken from x_i without compromising the security of the generator.

5   Assuming we only take 1 bit per x_i, N bits (and hence N iterations of the bit generator function) are needed in order to generate an N-bit random number. To the outside observer, given a particular set of bits, there is no way to determine the next bit other than a 50/50 probability. If the x, p and q are hidden, they act as a key, and it is computationally infeasible to take an output bit stream and compute x, p, and q. It is also computationally infeasible to determine the value of i used to

10   generate a given set of pseudo-random bits. This last feature makes the generator one-way. Different values of i can produce identical bit sequences of a given length (e.g. 32 bits of random bits). Even if x, p and q are known, for a given F[i], i can only be derived as a set of possibilities, not as a certain value (of course if the domain of i is known, then the set of possibilities is reduced further).

15

However, there are problems in selecting a good p and q, and a good seed x. In particular, Ritter in [68] describes a problem in selecting x. The nature of the problem is that a BBS generator does not create a single cycle of known length. Instead, it creates cycles of various lengths, including degenerate (zero-length) cycles. Thus a BBS generator cannot be initialized with a random state - it

20   might be on a short cycle. Specific algorithms exist in section 9 of [10] to determine the length of the period for a given seed given certain strenuous conditions for n.

5.5.3   Hash functions

Special one-way functions, known as Hash functions, map arbitrary length messages to fixed-

25   length hash values. Hash functions are referred to as H[M]. Since the input is of arbitrary length, a hash function has a compression component in order to produce a fixed length output. Hash functions also have an obfuscation component in order to make it difficult to find collisions and to determine information about M from H[M].

30   Because collisions do exist, most applications require that the hash algorithm is preimage resistant, in that for a given X_1 it is difficult to find X_2 such that H[X_1] = H[X_2]. In addition, most applications also require the hash algorithm to be *collision resistant* (i.e. it should be hard to find two messages X_1 and X_2 such that H[X_1] = H[X_2]). However, as described in [20], it is an open problem whether a collision-resistant hash function, in the ideal sense, can exist at all.

35

The primary application for hash functions is in the reduction of an input message into a digital "fingerprint" before the application of a digital signature algorithm. One problem of collisions with digital signatures can be seen in the following example.

5   A has a long message M_1 that says *"I owe B $10"*. A signs $H[M_1]$ using his private key. B, being greedy, then searches for a collision message M_2 where $H[M_2] = H[M_1]$ but where M_2 is favorable to B, for example *"I owe B $1million"*. Clearly it is in A's interest to ensure that it is difficult to find such an M_2.

10   Examples of collision resistant one-way hash functions are SHA-1 [28], MD5 [73] and RIPEMD-160 [66], all derived from MD4 [70][70].

5.5.3.1   MD4

Ron Rivest introduced MD4 [70][70] in 1990. It is only mentioned here because all other one-way hash functions are derived in some way from MD4.

15

MD4 is now considered completely broken [18][18] in that collisions can be calculated instead of searched for. In the example above, B could trivially generate a substitute message M_2 with the same hash value as the original message M_1.

20   ### 5.5.3.2   MD5

Ron Rivest introduced MD5 [73] in 1991 as a more secure MD4. Like MD4, MD5 produces a 128-bit hash value. MD5 is not patented [80].

Dobbertin describes the status of MD5 after recent attacks [20]. He describes how pseudo-
25   collisions have been found in MD5, indicating a weakness in the compression function, and more recently, collisions have been found. This means that MD5 should not be used for compression in digital signature schemes where the existence of collisions may have dire consequences. However MD5 can still be used as a one-way function. In addition, the HMAC-MD5 construct (see Section 5.5.4.1 on page 643) is not affected by these recent attacks.
30

5.5.3.3   SHA-1

SHA-1 [28] is very similar to MD5, but has a 160-bit hash value (MD5 only has 128 bits of hash value). SHA-1 was designed and introduced by the NIST and NSA for use in the Digital Signature Standard (DSS). The original published description was called SHA [27], but very soon afterwards,
35   was revised to become SHA-1 [28], supposedly to correct a security flaw in SHA (although the NSA has not released the mathematical reasoning behind the change).

There are no known cryptographic attacks against SHA-1 [78]. It is also more resistant to brute force attacks than MD4 or MD5 simply because of the longer hash result.

The US Government owns the SHA-1 and DSA algorithms (a digital signature authentication algorithm defined as part of DSS [29]) and has at least one relevant patent (US patent 5,231,688 granted in 1993). However, according to NIST [61]:

> "The DSA patent and any foreign counterparts that may issue are available for use
> without any written permission from or any payment of royalties to the U.S.
> government."

In a much stronger declaration, NIST states in the same document [61] that DSA and SHA-1 do not infringe third party's rights:

> "NIST reviewed all of the asserted patents and concluded that none of them would
> be infringed by DSS. Extra protection will be written into the PK1 pilot project that will
> prevent an organization or individual from suing anyone except the government for
> patent infringement during the course of the project."

It must however, be noted that the Schnorr authentication algorithm [81] (US patent number 4,995,082) patent holder claims that DSA infringes his patent. The Schnorr patent is not due to expire until 2008. Fortunately this does not affect SHA-1.

5.5.3.4   RIPEMD-160

RIPEMD-160 [66] is a hash function derived from its predecessor RIPEMD [11] (developed for the European Community's RIPE project in 1992). As its name suggests, RIPEMD-160 produces a 160-bit hash result. Tuned for software implementations on 32-bit architectures, RIPEMD-160 is intended to provide a high level of security for 10 years or more.

Although there have been no successful attacks on RIPEMD-160, it is comparatively new and has not been extensively cryptanalyzed. The original RIPEMD algorithm [11] was specifically designed to resist known cryptographic attacks on MD4. The recent attacks on MD5 (detailed in [20]) showed similar weaknesses in the RIPEMD 128-bit hash function. Although the attacks showed only theoretical weaknesses, Dobbertin, Preneel and Bosselaers further strengthened RIPEMD into a new algorithm RIPEMD-160.

RIPEMD-160 is in the public domain, and requires no licensing or royalty payments.

5.5.4   Message authentication codes

The problem of message authentication can be summed up as follows:

How can A be sure that a message supposedly from B is in fact from B?

Message authentication is different from entity authentication (described in the section on cryptographic challenge-response protocols). With entity authentication, one entity (the claimant) proves its identity to another (the verifier). With message authentication, we are concerned with making sure that a given message is from who we think it is from i.e. it has not been tampered with en route from the source to its destination. While this section has a brief overview of message authentication, a more detailed survey can be found in [88].

A one-way hash function is not sufficient protection for a message. Hash functions such as MD5 rely on generating a hash value that is representative of the original input, and the original input cannot be derived from the hash value. A simple attack by E, who is in-between A and B, is to intercept the message from B, and substitute his own. Even if A also sends a hash of the original message, E can simply substitute the hash of his new message. Using a one-way hash function alone, A has no way of knowing that B's message has been changed.

One solution to the problem of message authentication is the Message Authentication Code, or MAC.

When B sends message M, it also sends MAC[M] so that the receiver will know that M is actually from B. For this to be possible, only B must be able to produce a MAC of M, and in addition, A should be able to verify M against MAC[M]. Notice that this is different from encryption of M - MACs are useful when M does not have to be secret.

The simplest method of constructing a MAC from a hash function is to encrypt the hash value with a symmetric algorithm:
1. Hash the input message H[M]
2. Encrypt the hash $E_K[H[M]]$

This is more secure than first encrypting the message and then hashing the encrypted message. Any symmetric or asymmetric cryptographic function can be used, with the appropriate advantages and disadvantage of each type described in Section 5.2 on page 629 and Section 5.3 on page 632.

However, there are advantages to using a *key-dependent one-way hash function* instead of techniques that use encryption (such as that shown above):
- Speed, because one-way hash functions in general work much faster than encryption;

- Message size, because $E_K[M]$ is at least the same size as M, while H[M] is a fixed size (usually considerably smaller than M);
- Hardware/software requirements - keyed one-way hash functions are typically far less complex than their encryption-based counterparts; and
5
- One-way hash function implementations are not considered to be encryption or decryption devices and therefore are not subject to US export controls.

It should be noted that hash functions were never originally designed to contain a key or to support message authentication. As a result, some ad hoc methods of using hash functions to perform message authentication, including various functions that concatenate messages with secret

10 prefixes, suffixes, or both have been proposed [56][56]. Most of these ad hoc methods have been successfully attacked by sophisticated means [42][42][42]. Additional MACs have been suggested based on XOR schemes [8] and Toeplitz matrices [49] (including the special case of LFSR-based (Linear Feed Shift Register) constructions).

15 *5.5.4.1   HMAC*

The HMAC construction [6][6] in particular is gaining acceptance as a solution for Internet message authentication security protocols. The HMAC construction acts as a wrapper, using the underlying hash function in a black-box way. Replacement of the hash function is straightforward if desired due to security or performance reasons. However, the major advantage of the HMAC construct is that it

20 can be proven secure provided the underlying hash function has some reasonable cryptographic strengths - that is, HMAC's strengths are directly connected to the strength of the hash function [6].

Since the HMAC construct is a wrapper, any iterative hash function can be used in an HMAC. Examples include HMAC-MD5, HMAC-SHA1, HMAC-RIPEMD160 etc.

25

Given the following definitions:
- H   = the hash function (e.g. MD5 or SHA-1)
- n   = number of bits output from H (e.g. 160 for SHA-1, 128 bits for MD5)
- M   = the data to which the MAC function is to be applied
30
- K   = the secret key shared by the two parties
- ipad   = 0x36 repeated 64 times
- opad   = 0x5C repeated 64 times

The HMAC algorithm is as follows:
35
1.   Extend K to 64 bytes by appending 0x00 bytes to the end of K
2.   XOR the 64 byte string created in (1) with ipad
3.   append data stream M to the 64 byte string created in (2)

4.   Apply H to the stream generated in (3)

5.   XOR the 64 byte string created in (1) with opad

6.   Append the H result from (4) to the 64 byte string resulting from (5)

7.   Apply H to the output of (6) and output the result

5

Thus:

$HMAC[M] = H[(K \oplus opad) \mid H[(K \oplus ipad) \mid M]]$

The recommended key length is at least n bits, although it should not be longer than 64 bytes (the

10   length of the hashing block). A key longer than n bits does not add to the security of the function.

HMAC optionally allows truncation of the final output e.g. truncation to 128 bits from 160 bits.

The HMAC designers' Request for Comments [51] was issued in 1997, one year after the algorithm

15   was first introduced. The designers claimed that the strongest known attack against HMAC is based
on the frequency of collisions for the hash function H (see Section 14.10 on page 700), and is
totally impractical for minimally reasonable hash functions:

As an example, if we consider a hash function like MD5 where the output length is
128 bits, the attacker needs to acquire the correct message authentication tags

20   *computed (with the same secret key K) on about 2^{64} known plaintexts. This would*
require the processing of at least 2^{64} blocks under H, an impossible task in any
realistic scenario (for a block length of 64 bytes this would take 250,000 years in a
continuous 1 Gbps link, and without changing the secret key K all this time). This
attack could become realistic only if serious flaws in the collision behavior of the

25   *function H are discovered (e.g. Collisions found after 2^{30} messages). Such a*
discovery would determine the immediate replacement of function H (the effects of
such a failure would be far more severe for the traditional uses of H in the context of
digital signatures, public key certificates etc).

30   Of course, if a 160-bit hash function is used, then 2^{64} should be replaced with 2^{80}.

This should be contrasted with a regular collision attack on cryptographic hash functions where no
secret key is involved and 2^{64} off-line parallelizable operations suffice to find collisions.

35   More recently, HMAC protocols with replay prevention components [62] have been defined in order
to prevent the capture and replay of any M, HMAC[M] combination within a given time period.

644

Finally, it should be noted that HMAC is in the public domain [50], and incurs no licensing fees. There are no known patents infringed by HMAC.

5.6 RANDOM NUMBERS AND TIME VARYING MESSAGES

5   The use of a random number generator as a one-way function has already been examined. However, random number generator theory is very much intertwined with cryptography, security, and authentication.

There are a large number of issues concerned with defining good random number generators.
10   Knuth, in [48] describes what makes a generator good (including statistical tests), and the general problems associated with constructing them. Moreau gives a high level survey of the current state of the field in [60].

One of the uses for random numbers is to ensure that messages vary over time. Consider a system
15   where A encrypts commands and sends them to B. If the encryption algorithm produces the same output for a given input, an attacker could simply record the messages and play them back to fool B. There is no need for the attacker to crack the encryption mechanism other than to know which message to play to B (while pretending to be A). Consequently messages often include a random number and a time stamp to ensure that the message (and hence its encrypted counterpart) varies
20   each time.

Random number generators are also often used to generate keys. Although Klapper has recently shown [45] that a family of secure feedback registers for the purposes of building key-streams *does* exist, he does not give any practical construction. It is therefore best to say at the moment that all
25   generators are insecure for this purpose. For example, the Berlekamp-Massey algorithm [54], is a classic attack on an LFSR random number generator. If the LFSR is of length n, then only $2n$ bits of the sequence suffice to determine the LFSR, compromising the key generator.

If, however, the only role of the random number generator is to make sure that messages vary over
30   time, the security of the generator and seed is not as important as it is for session key generation. If however, the random number seed generator is compromised, and an attacker is able to calculate future "random" numbers, it can leave some protocols open to attack. Any new protocol should be examined with respect to this situation.

35   The actual type of random number generator required will depend upon the implementation and the purposes for which the generator is used. Generators include Blum, Blum, and Shub [10], stream ciphers such as RC4 by Ron Rivest [71], hash functions such as SHA-1 [28] and RIPEMD-160 [66],

645

and traditional generators such LFSRs (Linear Feedback Shift Registers) [48] and their more recent counterpart FCSRs (Feedback with Carry Shift Registers) [44].

5.7   ATTACKS

This section describes the various types of attacks that can be undertaken to break an authentication cryptosystem. The attacks are grouped into *physical* and *logical* attacks.

Logical attacks work on the protocols or algorithms rather than their physical implementation, and attempt to do one of three things:

- Bypass the authentication process altogether
- Obtain the secret key by force or deduction, so that *any* question can be answered
- Find enough about the nature of the authenticating questions and answers in order to, *without the key*, give the right answer to each question.

Regardless of the algorithms and protocol used by a security chip, the circuitry of the authentication part of the chip can come under physical attack. Physical attacks come in four main ways, although the form of the attack can vary:

- Bypassing the security chip altogether
- Physical examination of the chip while in operation (destructive and non-destructive)
- Physical decomposition of chip
- Physical alteration of chip

The attack styles and the forms they take are detailed below.

This section does not suggest solutions to these attacks. It merely describes each attack type. The examination is restricted to the context of an authentication chip (as opposed to some other kind of system, such as Internet authentication) attached to some System.

5.7.1   Logical attacks

These attacks are those which do not depend on the physical implementation of the cryptosystem. They work against the protocols and the security of the algorithms and random number generators.

5.7.1.1   Ciphertext only attack

This is where an attacker has one or more encrypted messages, all encrypted using the same algorithm. The aim of the attacker is to obtain the plaintext messages from the encrypted messages. Ideally, the key can be recovered so that all messages in the future can also be recovered.

646

5.7.1.2 Known plaintext attack

This is where an attacker has both the plaintext and the encrypted form of the plaintext. In the case of an authentication chip, a known-plaintext attack is one where the attacker can see the data flow between the system and the authentication chip. The inputs and outputs are observed (not chosen by the attacker), and can be analyzed for weaknesses (such as birthday attacks or by a search for differentially interesting input/output pairs).

A known plaintext attack can be carried out by connecting a logic analyzer to the connection between the system and the authentication chip.

5.7.1.3 Chosen plaintext attacks

A chosen plaintext attack describes one where a cryptanalyst has the ability to send any chosen message to the cryptosystem, and observe the response. If the cryptanalyst knows the algorithm, there may be a relationship between inputs and outputs that can be exploited by feeding a specific output to the input of another function.

The chosen plaintext attack is much stronger than the known plaintext attack since the attacker can choose the messages rather than simply observe the data flow.

On a system using an embedded authentication chip, it is generally very difficult to prevent chosen plaintext attacks since the cryptanalyst can logically pretend he/she is the system, and thus send any chosen bit-pattern streams to the authentication chip.

5.7.1.4 Adaptive chosen plaintext attacks

This type of attack is similar to the chosen plaintext attacks except that the attacker has the added ability to modify subsequent chosen plaintexts based upon the results of previous experiments. This is certainly the case with any system / authentication chip scenario described for consumables such as photocopiers and toner cartridges, especially since both systems and consumables are made available to the public.

5.7.1.5 Brute force attack

A *guaranteed* way to break *any* key-based cryptosystem algorithm is simply to try every key. Eventually the right one will be found. This is known as a *brute force attack*. However, the more key possibilities there are, the more keys must be tried, and hence the longer it takes (on average) to find the right one. If there are N keys, it will take a maximum of N tries. If the key is N bits long, it will take a maximum of 2^N tries, with a 50% chance of finding the key after only half the attempts

(2^{N-1}). The longer N becomes, the longer it will take to find the key, and hence the more secure the key is. Of course, an attack may guess the key on the first try, but this is more unlikely the longer the key is.

5   Consider a key length of 56 bits. In the worst case, all 2^{56} tests (7.2×10^{16} tests) must be made to find the key. In 1977, Diffie and Hellman described a specialized machine for cracking DES, consisting of one million processors, each capable of running one million tests per second [17]. Such a machine would take 20 hours to break any DES code.

10   Consider a key length of 128 bits. In the worst case, all 2^{128} tests (3.4×10^{38} tests) must be made to find the key. This would take ten billion years on an array of a trillion processors each running 1 billion tests per second.

With a long enough key length, a brute force attack takes too long to be worth the attacker's efforts.
15

5.7.1.6 Guessing attack

This type of attack is where an attacker attempts to simply "guess" the key. As an attack it is identical to the brute force attack (see Section 5.7.1.5 on page 647) where the odds of success depend on the length of the key.
20

5.7.1.7 Quantum computer attack

To break an n-bit key, a quantum computer [83] (NMR, Optical, or Caged Atom) containing n qubits embedded in an appropriate algorithm must be built. The quantum computer effectively exists in 2^n simultaneous coherent states. The trick is to extract the right coherent state without causing any
25   decoherence. To date this has been achieved with a 2 qubit system (which exists in 4 coherent states). It is thought possible to extend this to 6 qubits (with 64 simultaneous coherent states) within a few years.

Unfortunately, every additional qubit halves the relative strength of the signal representing the key.
30   This rapidly becomes a serious impediment to key retrieval, especially with the long keys used in cryptographically secure systems.

As a result, attacks on a cryptographically secure key (e.g. 160 bits) using a Quantum Computer are likely not to be feasible and it is extremely unlikely that quantum computers will have achieved
35   more than 50 or so qubits within the commercial lifetime of the authentication chips. Even using a 50 qubit quantum computer, 2^{110} tests are required to crack a 160 bit key.

5.7.1.8 Purposeful error attack

With certain algorithms, attackers can gather valuable information from the results of a bad input. This can range from the error message text to the time taken for the error to be generated.

5 A simple example is that of a userid/password scheme. If the error message usually says "Bad userid", then when an attacker gets a message saying "Bad password" instead, then they know that the userid is correct. If the message always says "Bad userid/password" then much less information is given to the attacker. A more complex example is that of the recent published method of cracking encryption codes from secure web sites [41]. The attack involves sending particular messages to a
10 server and observing the error message responses. The responses give enough information to learn the keys - even the lack of a response gives some information.

An example of algorithmic time can be seen with an algorithm that returns an error as soon as an erroneous bit is detected in the input message. Depending on hardware implementation, it may be
15 a simple method for the attacker to time the response and alter each bit one by one depending on the time taken for the error response, and thus obtain the key. Certainly in a chip implementation the time taken can be observed with far greater accuracy than over the Internet.

5.7.1.9 Birthday attack

20 This attack is named after the famous "birthday paradox" (which is not actually a paradox at all). The odds of one person sharing a birthday with another, is 1 in 365 (not counting leap years). Therefore there must be 183 people in a room for the odds to be more than 50% that one of them shares your birthday. However, there only needs to be 23 people in a room for there to be more than a 50% chance that any two share a birthday, as shown in the following relation:
25

$$Prob = 1 - \frac{nPr}{n^r} = 1 - \frac{365P23}{365^{23}} \approx 0.507$$

Birthday attacks are common attacks against hashing algorithms, especially those algorithms that combine hashing with digital signatures.

30 If a message has been generated and already signed, an attacker must search for a collision message that hashes to the same value (analogous to finding one person who shares your birthday). However, if the attacker can generate the message, the birthday attack comes into play. The attacker searches for two messages that share the same hash value (analogous to any two people sharing a birthday), only one message is acceptable to the person signing it, and the other
35 is beneficial for the attacker. Once the person has signed the original message the attacker simply

claims now that the person signed the alternative message - mathematically there is no way to tell which message was the original, since they both hash to the same value.

Assuming a brute force attack is the only way to determine a match, the weakening of an n-bit key by the birthday attack is $2^{n/2}$. A key length of 128 bits that is susceptible to the birthday attack has an effective length of only 64 bits.

5.7.1.10 Chaining attack

These are attacks made against the chaining nature of hash functions. They focus on the compression function of a hash function. The idea is based on the fact that a hash function generally takes arbitrary length input and produces a constant length output by processing the input n bits at a time. The output from one block is used as the chaining variable set into the next block. Rather than finding a collision against an entire input, the idea is that given an input chaining variable set, to find a substitute block that will result in the same output chaining variables as the proper message.

The number of choices for a particular block is based on the length of the block. If the chaining variable is c bits, the hashing function behaves like a random mapping, and the block length is b bits, the number of such b-bit blocks is approximately $2^b / 2^c$. The challenge for finding a substitution block is that such blocks are a sparse subset of all possible blocks.

For SHA-1, the number of 512 bit blocks is approximately $2^{512}/2^{160}$, or 2^{352}. The chance of finding a block by brute force search is about 1 in 2^{160}.

5.7.1.11 Substitution with a complete lookup table

If the number of potential messages sent to the chip is small, then there is no need for a clone manufacturer to crack the key. Instead, the clone manufacturer could incorporate a ROM in their chip that had a record of all of the responses from a genuine chip to the codes sent by the system. The larger the key, and the larger the response, the more space is required for such a lookup table.

5.7.1.12 Substitution with a sparse lookup table

If the messages sent to the chip are somehow predictable, rather than effectively random, then the clone manufacturer need not provide a complete lookup table. For example:

• If the message is simply a serial number, the clone manufacturer need simply provide a lookup table that contains values for past and predicted future serial numbers. There are unlikely to be more than 10^9 of these.

650

- If the test code is simply the date, then the clone manufacturer can produce a lookup table using the date as the address.

- If the test code is a pseudo-random number using either the serial number or the date as a seed, then the clone manufacturer just needs to crack the pseudo-random number generator in the system. This is probably not difficult, as they have access to the object code of the system. The clone manufacturer would then produce a content addressable memory (or other sparse array lookup) using these codes to access stored authentication codes.

5.7.1.13 Differential cryptanalysis

Differential cryptanalysis describes an attack where pairs of input streams are generated with known differences, and the differences in the encoded streams are analyzed.

Existing differential attacks are heavily dependent on the structure of S boxes, as used in DES and other similar algorithms. Although other algorithms such as HMAC-SHA1 have no S boxes, an attacker can undertake a differential-like attack by undertaking statistical analysis of:

- Minimal-difference inputs, and their corresponding outputs
- Minimal-difference outputs, and their corresponding inputs

Most algorithms were strengthened against differential cryptanalysis once the process was described. This is covered in the specific sections devoted to each cryptographic algorithm. However some recent algorithms developed in secret have been broken because the developers had not considered certain styles of differential attacks [94] and did not subject their algorithms to public scrutiny.

5.7.1.14 Message substitution attacks

In certain protocols, a man-in-the-middle can substitute part or all of a message. This is where a real authentication chip is plugged into a reusable clone chip within the consumable. The clone chip intercepts all messages between the system and the authentication chip, and can perform a number of substitution attacks.

Consider a message containing a header followed by content. An attacker may not be able to generate a valid header, but may be able to substitute their own content, especially if the valid response is something along the lines of "Yes, I received your message". Even if the return message is "Yes, I received the following message ...", the attacker may be able to substitute the original message before sending the acknowledgment back to the original sender.

Message Authentication Codes were developed to combat message substitution attacks.

5.7.1.15 Reverse engineering the key generator

If a pseudo-random number generator is used to generate keys, there is the potential for a clone manufacture to obtain the generator program or to deduce the random seed used. This was the way in which the security layer of the Netscape browser program was initially broken [33].

5.7.1.16 Bypassing the authentication process

It may be that there are problems in the authentication protocols that can allow a bypass of the authentication process altogether. With these kinds of attacks the key is completely irrelevant, and the attacker has no need to recover it or deduce it.

Consider an example of a system that authenticates at power-up, but does not authenticate at any other time. A reusable consumable with a clone authentication chip may make use of a real authentication chip. The clone authentication chip uses the real chip for the authentication call, and then simulates the real authentication chip's state data after that.

Another example of bypassing authentication is if the system authenticates only after the consumable has been used. A clone authentication chip can accomplish a simple authentication bypass by simulating a loss of connection after the use of the consumable but before the authentication protocol has completed (or even started).

One infamous attack known as the "Kentucky Fried Chip" hack [2] involved replacing a microcontroller chip for a satellite TV system. When a subscriber stopped paying the subscription fee, the system would send out a "disable" message. However the new micro-controller would simply detect this message and not pass it on to the consumer's satellite TV system.

5.7.1.17 Garrote/bribe attack

If people know the key, there is the possibility that they could tell someone else. The telling may be due to coercion (bribe, garrote etc.), revenge (e.g. a disgruntled employee), or simply for principle. These attacks are usually cheaper and easier than other efforts at deducing the key. As an example, a number of people claiming to be involved with the development of the (now defunct) Divx standard for DVD claimed (before the standard was rejected by consumers) that they would like to help develop Divx specific cracking devices - out of principle.

5.7.2   Physical attacks

The following attacks assume implementation of an authentication mechanism in a silicon chip that the attacker has physical access to. The first attack, *Reading ROM*, describes an attack when keys

are stored in ROM, while the remaining attacks assume that a secret key is stored in Flash memory.

5.7.2.1 Reading ROM

5 If a key is stored in ROM it can be read directly. A ROM can thus be safely used to hold a public key (for use in asymmetric cryptography), but not to hold a private key. In symmetric cryptography, a ROM is completely insecure. Using a copyright text (such as a *haiku*) as the key is not sufficient, because we are assuming that the cloning of the chip is occurring in a country where intellectual property is not respected.

10

5.7.2.2 Reverse engineering of chip

Reverse engineering of the chip is where an attacker opens the chip and analyzes the circuitry. Once the circuitry has been analyzed the inner workings of the chip's algorithm can be recovered. Lucent Technologies have developed an active method [4] known as TOBIC (Two photon OBIC,

15 where OBIC stands for Optical Beam Induced Current), to image circuits. Developed primarily for static RAM analysis, the process involves removing any back materials, polishing the back surface to a mirror finish, and then focusing light on the surface. The excitation wavelength is specifically chosen not to induce a current in the IC.

20 A Kerckhoffs in the nineteenth century made a fundamental assumption about cryptanalysis: *if the algorithm's inner workings are the sole secret of the scheme, the scheme is as good as broken* [39]. He stipulated that the secrecy must reside entirely in the key. As a result, the best way to protect against reverse engineering of the chip is to make the inner workings irrelevant.

25 ### 5.7.2.3 Usurping the authentication process

It must be assumed that any clone manufacturer has access to both the system and consumable designs.

If the same channel is used for communication between the system and a trusted system

30 authentication chip, and a non-trusted consumable authentication chip, it may be possible for the non-trusted chip to interrogate a trusted authentication chip in order to obtain the "correct answer". If this is so, a clone manufacturer would not have to determine the key. They would only have to trick the system into using the responses from the system authentication chip.

35 The alternative method of usurping the authentication process follows the same method as the logical attack described in Section 5.7.1.16 on page 652, involving simulated loss of contact with the system whenever authentication processes take place, simulating power-down etc.

653

5.7.2.4 Modification of system

This kind of attack is where the system itself is modified to accept clone consumables. The attack may be a change of system ROM, a rewiring of the consumable, or, taken to the extreme case, a completely clone system.

Note that this kind of attack requires each individual system to be modified, and would most likely require the owner's consent. There would usually have to be a clear advantage for the consumer to undertake such a modification, since it would typically void warranty and would most likely be costly. An example of such a modification with a clear advantage to the consumer is a software patch to change fixed-region DVD players into region-free DVD players (although it should be noted that this is not to use clone consumables, but rather originals from the same companies simply targeted for sale in other countries).

5.7.2.5 Direct viewing of chip operation by conventional probing

If chip operation could be directly viewed using an STM (Scanning Tunnelling Microscope) or an electron beam, the keys could be recorded as they are read from the internal non-volatile memory and loaded into work registers.

These forms of conventional probing require direct access to the top or front sides of the IC while it is powered.

5.7.2.6 Direct viewing of the non-volatile memory

If the chip were sliced so that the floating gates of the Flash memory were exposed, without discharging them, then the key could probably be viewed directly using an STM or SKM (Scanning Kelvin Microscope).

However, slicing the chip to this level without discharging the gates is probably impossible. Using wet etching, plasma etching, ion milling (focused ion beam etching), or chemical mechanical polishing will almost certainly discharge the small charges present on the floating gates.

5.7.2.7 Viewing the light bursts caused by state changes

Whenever a gate changes state, a small amount of infrared energy is emitted. Since silicon is transparent to infrared, these changes can be observed by looking at the circuitry from the underside of a chip. While the emission process is weak, it is bright enough to be detected by highly sensitive equipment developed for use in astronomy. The technique [92], developed by IBM, is called PICA (Picosecond Imaging Circuit Analyzer). If the state of a register is known at time t, then

watching that register change over time will reveal the exact value at time $t+n$, and if the data is part of the key, then that part is compromised.

5.7.2.8   Viewing the keys using an SEPM

A non-invasive testing device, known as a Scanning Electric Potential Microscope (SEPM), allows the direct viewing of charges within a chip [37]. The SEPM has a tungsten probe that is placed a few micrometers above the chip, with the probe and circuit forming a capacitor. Any AC signal flowing beneath the probe causes displacement current to flow through this capacitor. Since the value of the current change depends on the amplitude and phase of the AC signal, the signal can be imaged. If the signal is part of the key, then that part is compromised.

5.7.2.9   Monitoring EMI

Whenever electronic circuitry operates, faint electromagnetic signals are given off. Relatively inexpensive equipment can monitor these signals and could give enough information to allow an attacker to deduce the keys.

5.7.2.10   Viewing I_{dd} fluctuations

Even if keys cannot be viewed, there is a fluctuation in current whenever registers change state. If there is a high enough signal to noise ratio, an attacker can monitor the difference in I_{dd} that may occur when programming over either a high or a low bit. The change in I_{dd} can reveal information about the key. Attacks such as these have already been used to break smart cards [46].

5.7.2.11   Differential Fault Analysis

This attack assumes introduction of a bit error by ionization, microwave radiation, or environmental stress. In most cases such an error is more likely to adversely affect the chip (e.g. cause the program code to crash) rather than cause beneficial changes which would reveal the key. Targeted faults such as ROM overwrite, gate destruction etc. are far more likely to produce useful results.

5.7.2.12   Clock glitch attacks

Chips are typically designed to properly operate within a certain clock speed range. Some attackers attempt to introduce faults in logic by running the chip at extremely high clock speeds or introduce a clock glitch at a particular time for a particular duration [1]. The idea is to create race conditions where the circuitry does not function properly. An example could be an AND gate that (because of race conditions) gates through $Input_1$ all the time instead of the AND of $Input_1$ and $Input_2$.

If an attacker knows the internal structure of the chip, they can attempt to introduce race conditions at the correct moment in the algorithm execution, thereby revealing information about the key (or in the worst case, the key itself).

5.7.2.13 Power supply attacks

Instead of creating a glitch in the clock signal, attackers can also produce glitches in the power supply where the power is increased or decreased to be outside the working operating voltage range. The net effect is the same as a clock glitch - introduction of error in the execution of a particular instruction. The idea is to stop the CPU from XORing the key, or from shifting the data one bit-position etc. Specific instructions are targeted so that information about the key is revealed.

5.7.2.14 Overwriting ROM

Single bits in a ROM can be overwritten using a laser cutter microscope [1], to either 1 or 0 depending on the sense of the logic. If the ROM contains instructions, it may be a simple matter for an attacker to change a conditional jump to a non-conditional jump, or perhaps change the destination of a register transfer. If the target instruction is chosen carefully, it may result in the key being revealed.

5.7.2.15 Modifying EEPROM/Flash

These attacks fall into two categories:

- those similar to the ROM attacks except that the laser cutter microscope technique can be used to both set *and* reset individual bits. This gives much greater scope in terms of modification of algorithms.
- Electron beam programming of floating gates. As described in [89] and [32], a focused electron beam can change a gate by depositing electrons onto it. Damage to the rest of the circuit can be avoided, as described in [31].

5.7.2.16 Gate destruction

Anderson and Kuhn described the rump session of the 1997 workshop on Fast Software Encryption [1], where Biham and Shamir presented an attack on DES. The attack was to use a laser cutter to destroy an individual gate in the hardware implementation of a known block cipher (DES). The net effect of the attack was to force a particular bit of a register to be "stuck". Biham and Shamir described the effect of forcing a particular register to be affected in this way - the least significant bit of the output from the round function is set to 0. Comparing the 6 least significant bits of the left half and the right half can recover several bits of the key. Damaging a number of chips in this way can reveal enough information about the key to make complete key recovery easy.

An encryption chip modified in this way will have the property that encryption and decryption will no longer be inverses.

5.7.2.17 Overwrite attacks

5   Instead of trying to read the Flash memory, an attacker may simply set a single bit by use of a laser cutter microscope. Although the attacker doesn't know the previous value, they know the new value. If the chip still works, the bit's original state must be the same as the new state. If the chip doesn't work any longer, the bit's original state must be the logical NOT of the current state. An attacker can perform this attack on each bit of the key and obtain the n-bit key using at most n chips
10   (if the new bit matched the old bit, a new chip is not required for determining the next bit).

5.7.2.18 Test circuitry attack

Most chips contain test circuitry specifically designed to check for manufacturing defects. This includes BIST (Built In Self Test) and scan paths. Quite often the scan paths and test circuitry
15   includes access and readout mechanisms for all the embedded latches. In some cases the test circuitry could potentially be used to give information about the contents of particular registers.

Test circuitry is often disabled once the chip has passed all manufacturing tests, in some cases by blowing a specific connection within the chip. A determined attacker, however, can reconnect the
20   test circuitry and hence enable it.

5.7.2.19 Memory remnants

Values remain in RAM long after the power has been removed [35], although they do not remain long enough to be considered non-volatile. An attacker can remove power once sensitive
25   information has been moved into RAM (for example working registers), and then attempt to read the value from RAM. This attack is most useful against security systems that have regular RAM chips. A classic example is cited by [1], where a security system was designed with an automatic power-shut-off that is triggered when the computer case is opened. The attacker was able to simply open the case, remove the RAM chips, and retrieve the key because the values persisted.
30

5.7.2.20 Chip theft attack

If there are a number of stages in the lifetime of an authentication chip, each of these stages must be examined in terms of ramifications for security should chips be stolen. For example, if information is programmed into the chip in stages, theft of a chip between stages may allow an
35   attacker to have access to key information or reduced efforts for attack. Similarly, if a chip is stolen directly after manufacture but before programming, does it give an attacker any logical or physical advantage?

5.7.2.21 Trojan horse attack

At some stage the authentication chips must be programmed with a secret key. Suppose an attacker builds a clone authentication chip and adds it to the pile of chips to be programmed. The attacker has especially built the clone chip so that it looks and behaves just like a real

5   authentication chip, but will give the key out to the attacker when a special attacker-known command is issued to the chip. Of course the attacker must have access to the chip after the programming has taken place, as well as physical access to add the Trojan horse authentication chip to the genuine chips.

10   6   Requirements

Existing solutions to the problem of authenticating consumables have typically relied on patents covering physical packaging. However this does not stop home refill operations or clone manufacture in countries with weak industrial property protection. Consequently a much higher level of protection is required.

15

The authentication mechanism is therefore built into an authentication chip that is embedded in the consumable and allows a system to authenticate that consumable securely and easily. Limiting ourselves to the system authenticating consumables (we don't consider the consumable authenticating the system), two levels of protection can be considered:

20

Presence Only Authentication:

> This is where only the presence of an authentication chip is tested. The authentication chip can be removed and used in other consumables as long as be used indefinitely.

25   Consumable Lifetime Authentication:

> This is where not only is the presence of the authentication chip tested for, but also the authentication chip must only last the lifetime of the consumable. For the chip to be re-used it must be completely erased and reprogrammed.

30   The two levels of protection address different requirements. We are primarily concerned with Consumable Lifetime authentication in order to prevent cloned versions of high volume consumables. In this case, each chip should hold secure state information about the consumable being authenticated. It should be noted that a Consumable Lifetime authentication chip could be used in any situation requiring a Presence Only authentication chip.

35

Requirements for authentication, data storage integrity and manufacture are considered separately. The following sections summarize requirements of each.

6.1   AUTHENTICATION

The authentication requirements for both Presence Only and Consumable Lifetime authentication are restricted to the case of a system authenticating a consumable. We do not consider bi-directional authentication where the consumable also authenticates the system. For example, it is not necessary for a valid toner cartridge to ensure it is being used in a valid photocopier.

For Presence Only authentication, we must be assured that an authentication chip is physically present. For Consumable Lifetime authentication we also need to be assured that state data actually came from the authentication chip, and that it has not been altered en route. These issues cannot be separated - data that has been altered has a new source, and if the source cannot be determined, the question of alteration cannot be settled.

It is not enough to provide an authentication method that is secret, relying on a home-brew security method that has not been scrutinized by security experts. The primary requirement therefore is to provide authentication by means that have withstood the scrutiny of experts.

The authentication scheme used by the authentication chip should be resistant to defeat by logical means. Logical types of attack are extensive, and attempt to do one of three things:
*   Bypass the authentication process altogether
*   Obtain the secret key by force or deduction, so that any question can be answered
*   Find enough about the nature of the authenticating questions and answers in order to, without the key, give the right answer to each question.

The logical attack styles and the forms they take are detailed in Section 5.7.1 on page 646.

The algorithm should have a flat keyspace, allowing any random bit string of the required length to be a possible key. There should be no weak keys.

6.2   DATA STORAGE INTEGRITY

Although authentication protocols take care of ensuring data integrity in communicated messages, data storage integrity is also required. Two kinds of data must be stored within the authentication chip:
*   Authentication data, such as secret keys
*   Consumable state data, such as serial numbers, and media remaining etc.

The access requirements of these two data types differ greatly. The authentication chip therefore requires a storage/access control mechanism that allows for the integrity requirements of each type.

5   6.2.1   Authentication data

Authentication data must remain confidential. It needs to be stored in the chip during a manufacturing/programming stage of the chip's life, but from then on must not be permitted to leave the chip. It must be resistant to being read from non-volatile memory. The authentication scheme is responsible for ensuring the key cannot be obtained by deduction, and the manufacturing process

10   is responsible for ensuring that the key cannot be obtained by physical means.

The size of the authentication data memory area must be large enough to hold the necessary keys and secret information as mandated by the authentication protocols.

15   6.2.2   Consumable state data

Consumable state data can be divided into the following types. Depending on the application, there will be different numbers of each of these types of data items.
- Read Only
- ReadWrite

20   - Decrement Only

Read Only data needs to be stored in the chip during a manufacturing/programming stage of the chip's life, but from then on should not be allowed to change. Examples of Read Only data items are consumable batch numbers and serial numbers.

25   ReadWrite data is changeable state information, for example, the last time the particular consumable was used. ReadWrite data items can be read and written an unlimited number of times during the lifetime of the consumable. They can be used to store any state information about the consumable. The only requirement for this data is that it needs to be kept in non-volatile memory. Since an attacker can obtain access to a system (which can write to

30   ReadWrite data), any attacker can potentially change data fields of this type. This data type should not be used for secret information, and must be considered insecure.

Decrement Only data is used to count down the availability of consumable resources. A photocopier's toner cartridge, for example, may store the amount of toner remaining as a Decrement Only data item. An ink cartridge for a color printer may store the amount of each

35   ink color as a Decrement Only data item, requiring 3 (one for each of Cyan, Magenta, and Yellow), or even as many as 5 or 6 Decrement Only data items. The requirement for this kind of data item is that once programmed with an initial value at the manufacturing/programming

stage, *it can only reduce in value*. Once it reaches the minimum value, it cannot decrement any further. The Decrement Only data item is only required by Consumable Lifetime authentication.

5   Note that the size of the consumable state data storage required is only for that information required to be authenticated. Information which would be of no use to an attacker, such as ink color-curve characteristics or ink viscosity do not have to be stored in the secure state data memory area of the authentication chip.

10   6.3   MANUFACTURE
The authentication chip must have a low manufacturing cost in order to be included as the authentication mechanism for low cost consumables.

The authentication chip should use a standard manufacturing process, such as Flash. This is
15   necessary to:
   • Allow a great range of manufacturing location options
   • Use well-defined and well-behaved technology
   • Reduce cost

20   Regardless of the authentication scheme used, the circuitry of the authentication part of the chip must be resistant to physical attack. Physical attack comes in four main ways, although the form of the attack can vary:
   • Bypassing the authentication chip altogether
   • Physical examination of chip while in operation (destructive and non-destructive)
25   • Physical decomposition of chip
   • Physical alteration of chip

The physical attack styles and the forms they take are detailed in Section 5.7.2 on page 652. Ideally, the chip should be exportable from the USA, so it should not be possible to use an
30   authentication chip as a secure encryption device. This is low priority requirement since there are many companies in other countries able to manufacture the authentication chips. In any case, the export restrictions from the USA may change.

AUTHENTICATION
35   7   Introduction
Existing solutions to the problem of authenticating consumables have typically relied on physical patents on packaging. However this does not stop home refill operations or clone manufacture in

661

countries with weak industrial property protection. Consequently a much higher level of protection is required.

It is not enough to provide an authentication method that is secret, relying on a home-brew security method that has not been scrutinized by security experts. Security systems such as Netscape's original proprietary system and the GSM Fraud Prevention Network used by cellular phones are examples where design secrecy caused the vulnerability of the security [33][33]. Both security systems were broken by conventional means that would have been detected if the companies had followed an open design process. The solution is to provide authentication by means that have withstood the scrutiny of experts.

In this section, we examine a number of protocols that can be used for consumables authentication. We only use security methods that are publicly described, using known behaviors in this new way. Readers should be familiar with the concepts and terms described in Section 5 on page 629. We avoid the Zero Knowledge Proof protocol since it is patented.

For all protocols, the security of the scheme relies on a secret key, not a secret algorithm. In the nineteenth century, A Kerckhoffs made a fundamental assumption about cryptanalysis: *if the algorithm's inner workings are the sole secret of the scheme, the scheme is as good as broken* [39]. He stipulated that the secrecy must reside entirely in the key. As a result, the best way to protect against reverse engineering of any authentication chip is to make the algorithmic inner workings irrelevant (the algorithm of the inner workings must still be must be valid, but not the actual secret).

The QA Chip is a programmable device, and can therefore be setup with an application-specific program together with an application-specific set of protocols. This section describes the following sets of protocols:
- single key single memory vector
- multiple key single memory vector
- multiple key multiple memory vector

These protocols refer to the number of valid keys that an QA Chip knows about, and the size of data required to be stored in the chip.

From these protocols it is straightforward to construct protocol sets for the single key multiple memory vector case (of course the multiple memory vector can be considered to be . and multiple key single memory vector. Other protocol sets can also be defined as necessary. Of course multiple memory vector can be conveniently

All the protocols rely on a time-variant challenge (i.e. the challenge is different each time), where the response depends on the challenge and the secret. The challenge involves a random number so that any observer will not be able to gather useful information about a subsequent identification.

8 Single Key Single Memory Vector

8.1 PROTOCOL BACKGROUND

This protocol set is provided for two reasons:

- the other protocol sets defined in this document are simply extensions of this one; and
- it is useful in its own right

The single key protocol set is useful for applications where only a single key is required. Note that there can be many consumables and systems, but there is only a single key that connects them all. Examples include:

- car and keys. A car and the car-key share a single key. There can be multiple sets of car-keys, each effectively cut to the same key. A company could have a set of cars, each with the same key. Any of the car-keys could then be used to drive any of the cars.
- printer and ink cartridge. All printers of a certain model use the same ink cartridge, with printer and cartridge sharing only a single key. Note that to introduce a new printer model that accepts the old ink cartridge the new model would need the same key as the old model. See the multiple-key protocols for alternative solutions to this problem.

8.2 REQUIREMENTS OF PROTOCOL

Each QA Chip contains the following values:

K	The secret key for calculating $F_K[X]$. K must not be stored directly in the QA Chip. Instead, each chip needs to store a random number R_K (different for each chip), $K \oplus R_K$, and $\neg K \oplus R_K$. The stored $K \oplus R_K$ can be XORed with R_K to obtain the real K. Although $\neg K \oplus R_K$ must be stored to protect against differential attacks, it is not used.
R	Current random number used to ensure time varying messages. Each chip instance must be seeded with a different initial value. Changes for each signature generation.
M	Memory vector of QA Chip.
P	2 element array of access permissions for each part of M. Entry 0 holds access permissions for non-authenticated writes to M (no key required). Entry 1 holds access permissions for authenticated writes to M (key required). Permission choices for each part of M are Read Only, Read/Write, and Decrement Only.

C	3 constants used for generating signatures. C_1, C_2, and C_3 are constants that pad out a submessage to a hashing boundary, and all 3 must be different.

Each QA Chip contains the following private function:

$S_K[X]$	*Internal function only.* Returns $S_K[X]$, the result of applying a digital signature function S to X based upon key K. The digital signature must be long enough to counter the chances of someone generating a random signature. The length depends on the signature scheme chosen, although the scheme chosen for the QA Chip is HMAC-SHA1 (see Section 13 on page 691), and therefore the length of the signature is 160 bits.

Additional functions are required in certain QA Chips, but these are described as required.

8.3   READS OF M

In this case, we have a trusted chip (*ChipT*) connected to a System. The System wants to authenticate an object that contains a non-trusted chip (*ChipA*). In effect, the System wants to know that it can securely read a memory vector (M) from ChipA: to be sure that ChipA is valid and that M has not been altered.

The protocol requires the following publicly available function in ChipA:

Read[X]	Advances R, and returns R, M, $S_K[X|R|C_1|M]$. The time taken to calculate the signature must not be based on the contents of X, R, M, or K.

The protocol requires the following publicly available functions in ChipT:

Random[]	Returns R (does not advance R).
Test[X, Y, Z]	Advances R and returns 1 if $S_K[R|X|C_1|Y] = Z$. Otherwise returns 0. The time taken to calculate and compare signatures must be independent of data content.

To authenticate ChipA and read ChipA's memory M:

a. System calls ChipT's Random function;

b. ChipT returns R_T to System;

c. System calls ChipA's Read function, passing in the result from b;

d. ChipA updates R_A, then calculates and returns R_A, M_A, $S_K[R_T|R_A|C_1|M_A]$;

e. System calls ChipT's Test function, passing in R_A, M_A, $S_K[R_T|R_A|C_1|M_A]$;

f. System checks response from ChipT. If the response is 1, then ChipA is considered authentic. If 0, ChipA is considered invalid.

The data flow for read authentication is shown in Figure 334.

The protocol allows System to simply pass data from one chip to another, with no special processing. The protection relies on ChipT being trusted, even though System does not know K.

5   When ChipT is physically separate from System (eg is chip on a board connected to System) System *must also occassionally* (based on system clock for example) call ChipT's Test function with bad data, expecting a 0 response. This is to prevent someone from inserting a fake ChipT into the system that always returns 1 for the Test function.

8.4   WRITES

10 In this case, the System wants to update M in some chip referred to as *ChipU*. This can be non-authenticated (for example, anyone is allowed to count down the amount of consumable remaining), or authenticated (for example, replenishing the amount of consumable remaining).

8.4.1   Non-authenticated writes

15 This is the most frequent type of write, and takes place between the System / consumable during normal everyday operation. In this kind of write, System wants to change M in a way that doesn't require special authorization. For example, the System could be decrementing the amount of consumable remaining. Although *System does not need to know K or even have access to a trusted chip*, System must follow a non-authenticated write by an authenticated read if it needs to

20 know that the write was successful.

The protocol requires the following publicly available function:

Write[X]   Writes X over those parts of M subject to P_0 and the existing value for M.

25 To authenticate a write of M_{new} to ChipA's memory M:

a. System calls ChipU's Write function, passing in M_{new};
b. The authentication procedure for a Read is carried out (see Section 8.3 on page 664);
c. If ChipU is authentic and M_{new} = M returned in b, the write succeeded. If not, it failed.

30 ### 8.4.2   Authenticated writes

In this kind of write, System wants to change Chip U's M in an authorized way, without being subject to the permissions that apply during normal operation (P_0). For example, the consumable may be at a refilling station and the normally Decrement Only section of M should be updated to include the new valid consumable. In this case, the chip whose M is being updated must

35 authenticate the writes being generated by the external System and in addition, apply permissions P_1 to ensure that only the correct parts of M are updated.

In this transaction protocol, the System's chip is referred to as ChipS, and the chip being updated is referred to as ChipU. Each chip distrusts the other.

The protocol requires the following publicly available functions in ChipU:

Read[X]   Advances R, and returns R, M, $S_K[X|R|C_1|M]$. The time taken to calculate the signature must be identical for all inputs.

WriteA[X, Y, Z]Returns 1, advances R, and replaces M by Y subject to P_1 only if $S_K[R|X|C_1|Y] = Z$. Otherwise returns 0. The time taken to calculate and compare signatures must be independent of data content. This function is identical to ChipT's Test function except that it additionally writes Y over those parts of M subject to P_1 when the signature matches.

Authenticated writes require that the System has access to a ChipS that is capable of generating appropriate signatures. ChipS requires the following variables and function:

CountRemaining   Part of M that contains the number of signatures that ChipS is allowed to generate. Decrements with each successful call to SignM and SignP. Permissions in ChipS's P_0 for this part of M needs to be ReadOnly once ChipS has been setup. Therefore CountRemaining can only be updated by another ChipS that will perform updates to that part of M (assuming ChipS's P_1 allows that part of M to be updated).

Q   Part of M that contains the write permissions for updating ChipU's M. By adding Q to ChipS we allow different ChipSs that can update different parts of M_U. Permissions in ChipS's P_0 for this part of M needs to be ReadOnly once ChipS has been setup. Therefore Q can only be updated by another ChipS that will perform updates to that part of M.

SignM[V,W,X,Y,Z]   Advances R, decrements CountRemaining and returns R, Z_{QX} (Z applied to X with permissions Q), followed by $S_K[W|R|C_1|Z_{QX}]$ only if $S_K[V|W|C_1|X] = Y$ and CountRemaining > 0. Otherwise returns all 0s. The time taken to calculate and compare signatures must be independent of data content.

To update ChipU's M vector:

a. System calls ChipU's Read function, passing in 0 as the input parameter;

b. ChipU produces R_U, M_U, $S_K[0|R_U|C_1|M_U]$ and returns these to System;

c. System calls ChipS's SignM function, passing in 0 (as used in a), R_U, M_U, $S_K[0|R_U|C_1|M_U]$, and M_D (the desired vector to be written to ChipU);

d. ChipS produces R_S, M_{QD} (processed by running M_D against M_U using Q) and $S_K[R_U|R_S|C_1|M_{QD}]$ if the inputs were valid, and 0 for all outputs if the inputs were not valid.

e. If values returned in d are non zero, then ChipU is considered authentic. System can then call ChipU's WriteA function with these values.

f. ChipU should return a 1 to indicate success. A 0 should only be returned if the data generated by ChipS is incorrect (e.g. a transmission error).

5   The data flow for authenticated writes is shown in Figure 335.

Note that Q in ChipS is part of ChipS's M. This allows a user to set up ChipS with a permission set for upgrades. This should be done to ChipS and that part of M designated by P_0 set to ReadOnly *before* ChipS is programmed with K_U. If K_S is programmed with K_U first, there is a risk of someone

10   obtaining a half-setup ChipS and changing all of M_U instead of only the sections specified by Q.

The same is true of CountRemaining. The CountRemaining value needs to be setup (including making it ReadOnly in P_0) before ChipS is programmed with K_U. ChipS is therefore programmed to only perform a limited number of SignM operations (thereby limiting compromise exposure if a

15   ChipS is stolen). Thus ChipS would itself need to be upgraded with a new CountRemaining every so often.

8.4.3   Updating permissions for future writes

In order to reduce exposure to accidental and malicious attacks on P and certain parts of M, only

20   authorized users are allowed to update P. Writes to P are the same as authorized writes to M, except that they update P_n instead of M. Initially (at manufacture), P is set to be Read/Write for all parts of M. As different processes fill up different parts of M, they can be sealed against future change by updating the permissions. Updating a chip's P_0 changes permissions for unauthorized writes, and updating P_1 changes permissions for authorized writes.

25

P_n is only allowed to change to be a more restrictive form of itself. For example, initially all parts of M have permissions of Read/Write. A permission of Read/Write can be updated to Decrement Only or Read Only. A permission of Decrement Only can be updated to become Read Only. A Read Only permission cannot be further restricted.

30

In this transaction protocol, the System's chip is referred to as ChipS, and the chip being updated is referred to as ChipU. Each chip distrusts the other.

The protocol requires the following publicly available functions in ChipU:

35   Random[]   Returns R (does not advance R).

SetPermission[n,X,Y,Z] Advances R, and updates P_n according to Y and returns 1 followed by the resultant P_n only if $S_K[R|X|Y|C_2] = Z$. Otherwise returns 0. P_n can only

become more restricted. Passing in 0 for any permission leaves it unchanged (passing in Y=0 returns the current P_n).

Authenticated writes of permissions require that the System has access to a ChipS that is capable of generating appropriate signatures. ChipS requires the following variables and function:

CountRemaining Part of M that contains the number of signatures that ChipS is allowed to generate. Decrements with each successful call to SignM and SignP. Permissions in ChipS's P_0 for this part of M needs to be ReadOnly once ChipS has been setup. Therefore CountRemaining can only be updated by another ChipS that will perform updates to that part of M (assuming ChipS's P_1 allows that part of M to be updated).

SignP[X,Y] Advances R, decrements CountRemaining and returns R and $S_K[X|R|Y|C_2]$ only if CountRemaining > 0. Otherwise returns all 0s. The time taken to calculate and compare signatures must be independent of data content.

To update ChipU's P_n:

a. System calls ChipU's Random function;

b. ChipU returns R_U to System;

c. System calls ChipS's SignP function, passing in R_U and P_D (the desired P to be written to ChipU);

d. ChipS produces R_S and $S_K[R_U|R_S|P_D|C_2]$ if it is still permitted to produce signatures.

e. If values returned in d are non zero, then System can then call ChipU's SetPermission function with the desired n, R_S, P_D and $S_K[R_U|R_S|P_D|C_2]$.

f. ChipU verifies the received signature against $S_K[R_U|R_S|P_D|C_2]$ and applies P_D to P_n if the signature matches

g. System checks 1st output parameter. 1 = success, 0 = failure.

The data flow for authenticated writes to permissions is shown in Figure 336 below.

8.5   PROGRAMMING K

In this case, we have a factory chip (*ChipF*) connected to a System. The System wants to program the key in another chip (*ChipP*). System wants to avoid passing the new key to ChipP in the clear, and also wants to avoid the possibility of the key-upgrade message being replayed on another ChipP (even if the user doesn't know the key).

The protocol assumes that ChipF and ChipP already share a secret key K_{old}. This key is used to ensure that only a chip that knows K_{old} can set K_{new}.

The protocol requires the following publicly available functions in ChipP:

Random[]   Returns R (does not advance R).

ReplaceKey[X, Y, Z]   Replaces K by $S_{Kold}[R|X|C_3] \oplus Y$, advances R, and returns 1 only if $S_{Kold}[X|Y|C_3] = Z$. Otherwise returns 0. The time taken to calculate signatures and compare values must be identical for all inputs.

And the following data and function in ChipF:

CountRemaining   Part of M with contains the number of signatures that ChipF is allowed to generate. Decrements with each successful call to GetProgramKey. Permissions in P for this part of M needs to be ReadOnly once ChipF has been setup. Therefore can only be updated by a ChipS that has authority to perform updates to that part of M.

K_{new} The new key to be transferred from ChipF to ChipP. Must not be visible.

SetPartialKey[X,Y]   If word X of K_{new} has not yet been set, set word X of K_{new} to Y and return 1. Otherwise return 0. This function allows K_{new} to be programmed in multiple steps, thereby allowing different people or systems to know different parts of the key (but not the whole K_{new}). K_{new} is stored in ChipF's flash memory. Since there is a small number of ChipFs, it is theoretically not necessary to store the inverse of K_{new}, but it is stronger protection to do so.

GetProgramKey[X]   Advances R_F, decrements CountRemaining, outputs R_F, the encrypted key $S_{Kold}[X|R_F|C_3] \oplus K_{new}$ and a signature of the first two outputs plus C_3 if CountRemaining>0. Otherwise outputs 0. The time to calculate the encrypted key & signature must be identical for all inputs.

To update P's key :

a. System calls ChipP's Random function;

b. ChipP returns R_P to System;

c. System calls ChipF's GetProgramKey function, passing in the result from b;

d. ChipF updates R_F, then calculates and returns R_F, $S_{Kold}[R_P|R_F|C_3] \oplus K_{new}$, and $S_{Kold}[R_F|S_{Kold}[R_P|R_F|C_3] \oplus K_{new}|C_3]$;

e. If the response from d is not 0, System calls ChipP's ReplaceKey function, passing in the response from d;

f. System checks response from ChipP. If the response is 1, then K_P has been correctly updated to K_{new}. If the response is 0, K_P has not been updated.

The data flow for key updates is shown in Figure 337.

Note that K_{new} is never passed in the open. An attacker could send its own R_P, but cannot produce $S_{Kold}[R_P|R_F|C_3]$ without K_{old}. The third parameter, a signature, is sent to ensure that ChipP can determine if either of the first two parameters have been changed en route.

5   CountRemaining needs to be setup in M_F (including making it ReadOnly in P) before ChipF is programmed with K_P. ChipF should therefore be programmed to only perform a limited number of GetProgramKey operations (thereby limiting compromise exposure if a ChipF is stolen). An authorized ChipS can be used to update this counter if neccesary (see Section 8.4 on page 665).

10   8.5.1   Chicken and Egg

Of course, for the Program Key protocol to work, both ChipF and ChipP must both know K_{old}. Obviously both chips had to be programmed with K_{old}, and thus K_{old} can be thought of as an older K_{new}: K_{old} can be placed in chips if another ChipF knows K_{older}, and so on.

15   Although this process allows a chain of reprogramming of keys, with each stage secure, at some stage the very first key (K_{first}) must be placed in the chips. K_{first} is in fact programmed with the chip's microcode at the manufacturing test station as the last step in manufacturing test. K_{first} can be a manufacturing batch key, changed for each batch or for each customer etc, and can have as short a life as desired. Compromising K_{first} need not result in a complete compromise of the chain of Ks.

20

9   Multiple Key Single Memory Vector

9.1   PROTOCOL BACKGROUND

This protocol set is an extension to the single key single memory vector protocol set, and is provided for two reasons:

25   •   the multiple key multiple memory vector protocol set defined in this document is simply extensions of this one; and

•   it is useful in its own right

The multiple key protocol set is typically useful for applications where there are multiple types of systems and consumables, and they need to work with each other in various ways. This is typically

30   in the following situations:

•   when different systems want to share some consumables, but not others. For example printer models may share some ink cartridges and not share others.

•   when there are different owners of data in M. Part of the memory vector may be owned by one company (eg the speed of the printer) and another may be owned by another (eg the

35   serial number of the chip). In this case a given key K_n needs to be able to write to a given part of M, and other keys K_n need to be disallowed from writing to these same areas.

9.2   REQUIREMENTS OF PROTOCOL

670

Each QA Chip contains the following values:

N	The maximum number of keys known to the chip.
K_N	Array of N secret keys used for calculating $F_{Kn}[X]$ where K_n is the nth element of the array. Each K_n must not be stored directly in the QA Chip . Instead, each chip needs to store a single random number R_K (different for each chip), $K_n \oplus R_K$, and $\neg K_n \oplus R_K$. The stored $K_n \oplus R_K$ can be XORed with R_K to obtain the real K_n. Although $\neg K_n \oplus R_K$ must be stored to protect against differential attacks, it is not used.
R	Current random number used to ensure time varying messages. Each chip instance must be seeded with a different initial value. Changes for each signature generation.
M	Memory vector of QA Chip. A fixed part of M contains N in ReadOnly form so users of the chip can know the number of keys known by the chip.
P	N+1 element array of access permissions for each part of M. Entry 0 holds access permissions for non-authenticated writes to M (no key required). Entries 1 to N+1 hold access permissions for authenticated writes to M, one for each K. Permission choices for each part of M are Read Only, Read/Write, and Decrement Only.
C	3 constants used for generating signatures. C_1, C_2, and C_3 are constants that pad out a submessage to a hashing boundary, and all 3 must be different.

Each QA Chip contains the following private function:

$S_{Kn}[N,X]$	*Internal function only.* Returns $S_{Kn}[X]$, the result of applying a digital signature function S to X based upon the appropriate key K_n. The digital signature must be long enough to counter the chances of someone generating a random signature. The length depends on the signature scheme chosen, although the scheme chosen for the QA Chip is HMAC-SHA1 (see Section 13 on page 691), and therefore the length of the signature is 160 bits.

Additional functions are required in certain QA Chips, but these are described as required.

9.3 READS

As with the single key scenario, we have a trusted chip (*ChipT*) connected to a System. The System wants to authenticate an object that contains a non-trusted chip (*ChipA*). In effect, the System wants to know that it can securely read a memory vector (M) from ChipA: to be sure that ChipA is valid and that M has not been altered.

The protocol requires the following publicly available functions:

Random[]	Returns R (does not advance R).
Read[n, X]	Advances R, and returns R, M, $S_{Kn}[X|R|C_1|M]$. The time taken to calculate the signature must not be based on the contents of X, R, M, or K.

Test[n,X, Y, Z]   Advances R and returns 1 if $S_{Kn}[R|X|C_1|Y] = Z$. Otherwise returns 0. The time taken to calculate and compare signatures must be independent of data content.

To authenticate ChipA and read ChipA's memory M:

5
a. System calls ChipT's Random function;

b. ChipT returns R_T to System;

c. System calls ChipA's Read function, passing in some key number n1 and the result from b;

d. ChipA updates R_A, then calculates and returns R_A, M_A, $S_{KAn1}[R_T|R_A|C_1|M_A]$;

e. System calls ChipT's Test function, passing in n2, R_A, M_A, $S_{KAn1}[R_T|R_A|C_1|M_A]$;

10
f. System checks response from ChipT. If the response is 1, then ChipA is considered authentic. If 0, ChipA is considered invalid.

The choice of n1 and n2 must be such that ChipA's K_{n1} = ChipT's K_{n2}.

15
The data flow for read authentication is shown in Figure 338.

The protocol allows System to simply pass data from one chip to another, with no special processing. The protection relies on ChipT being trusted, even though System does not know K.

20
When ChipT is physically separate from System (eg is chip on a board connected to System) System *must also occassionally* (based on system clock for example) call ChipT's Test function with bad data, expecting a 0 response. This is to prevent someone from inserting a fake ChipT into the system that always returns 1 for the Test function.

25
It is important that n1 is chosen by System. Otherwise ChipA would need to return N_A sets of signatures for each read, since ChipA does not know which of the keys will satisfy ChipT. Similarly, system must also choose n2, so it can potentially restrict the number of keys in ChipT that are matched against (otherwise ChipT would have to match against all its keys). This is important in order to restrict how different keys are used. For example, say that ChipT contains 6 keys, keys 0-2 are for various printer-related upgrades, and keys 3-6 are for inks. ChipA contains say 4 keys, one
30
key for each printer model. At power-up, System goes through each of chipA's keys 0-3, trying each out against ChipT's keys 3-6. System doesn't try to match against ChipT's keys 0-2. Otherwise knowledge of a speed-upgrade key could be used to provide ink QA Chip chips. This matching needs to be done only once (eg at power up). Once matching keys are found, System can continue
35
to use those key numbers.

Since System needs to know N_T and N_A, part of M is used to hold N (eg in Read Only form), and the system can obtain it by calling the Read function, passing in key 0.

9.4 WRITES

As with the single key scenario, the System wants to update M in ChipU. As before, this can be done in a non-authenticated and authenticated way.

9.4.1 Non-authenticated writes

This is the most frequent type of write, and takes place between the System / consumable during normal everyday operation. In this kind of write, System wants to change M subject to P. For example, the System could be decrementing the amount of consumable remaining. Although *System does not need to know any of the Ks or even have access to a trusted chip* to perform the write, System must follow a non-authenticated write by an authenticated read if it needs to know that the write was successful.

The protocol requires the following publicly available function:

Write[X]   Writes X over those parts of M subject to P_0 and the existing value for M.

To authenticate a write of M_{new} to ChipA's memory M:

a. System calls ChipU's Write function, passing in M_{new};

b. The authentication procedure for a Read is carried out (see Section 9.3 on page 671);

c. If ChipU is authentic and M_{new} = M returned in b, the write succeeded. If not, it failed.

9.4.2 Authenticated writes

In this kind of write, System wants to change Chip U's M in an authorized way, without being subject to the permissions that apply during normal operation (P_0). For example, the consumable may be at a refilling station and the normally Decrement Only section of M should be updated to include the new valid consumable. In this case, the chip whose M is being updated must authenticate the writes being generated by the external System and in addition, apply the appropriate permission for the key to ensure that only the correct parts of M are updated. Having a different permission for each key is required as when multiple keys are involved, all keys should not necessarily be given open access to M. For example, suppose M contains printer speed and a counter of money available for franking. A ChipS that updates printer speed should not be capable of updating the amount of money. Since P_0 is used for non-authenticated writes, each K_n has a corresponding permission P_{n+1} that determines what can be updated in an authenticated write.

In this transaction protocol, the System's chip is referred to as ChipS, and the chip being updated is referred to as ChipU. Each chip distrusts the other.

673

The protocol requires the following publicly available functions in ChipU:

Read[n, X] Advances R, and returns R, M, $S_{Kn}[X|R|C_1|M]$. The time taken to calculate the signature must not be based on the contents of X, R, M, or K.

WriteA[n, X, Y, Z] Advances R, replaces M by Y subject to P_{n+1}, and returns 1 only if $S_{Kn}[R|X|C_1|Y] = Z$. Otherwise returns 0. The time taken to calculate and compare signatures must be independent of data content. This function is identical to ChipT's Test function except that it additionally writes Y subject to P_{n+1} to its M when the signature matches.

Authenticated writes require that the System has access to a ChipS that is capable of generating appropriate signatures. ChipS requires the following variables and function:

CountRemaining Part of M that contains the number of signatures that ChipS is allowed to generate. Decrements with each successful call to SignM and SignP. Permissions in ChipS's P_0 for this part of M needs to be ReadOnly once ChipS has been setup. Therefore CountRemaining can only be updated by another ChipS that will perform updates to that part of M (assuming ChipS's P allows that part of M to be updated).

Q Part of M that contains the write permissions for updating ChipU's M. By adding Q to ChipS we allow different ChipSs that can update different parts of M_U. Permissions in ChipS's P_0 for this part of M needs to be ReadOnly once ChipS has been setup. Therefore Q can only be updated by another ChipS that will perform updates to that part of M.

SignM[n,V,W,X,Y,Z] Advances R, decrements CountRemaining and returns R, Z_{QX} (Z applied to X with permissions Q), $S_{Kn}[W|R|C_1|Z_{QX}]$ only if $Y = S_{Kn}[V|W|C_1|X]$ and CountRemaining > 0. Otherwise returns all 0s. The time taken to calculate and compare signatures must be independent of data content.

To update ChipU's M vector:
a. System calls ChipU's Read function, passing in n1 and 0 as the input parameters;
b. ChipU produces R_U, M_U, $S_{Kn1}[0|R_U|C_1|M_U]$ and returns these to System;
c. System calls ChipS's SignM function, passing in n2 (the key to be used in ChipS), 0 (as used in a), R_U, M_U, $S_{Kn1}[0|R_U|C_1|M_U]$, and M_D (the desired vector to be written to ChipU);
d. ChipS produces R_S, M_{QD} (processed by running M_D against M_U using Q) and $S_{Kn2}[R_U|R_S|C_1|M_{QD}]$ if the inputs were valid, and 0 for all outputs if the inputs were not valid.
e. If values returned in d are non zero, then ChipU is considered authentic. System can then call ChipU's WriteA function with these values from d.
f. ChipU should return a 1 to indicate success. A 0 should only be returned if the data generated by ChipS is incorrect (e.g. a transmission error).

674

The choice of n1 and n2 must be such that ChipU's K_{n1} = ChipS's K_{n2}.

The data flow for authenticated writes is shown in Figure 339 below.

5   Note that Q in ChipS is part of ChipS's M. This allows a user to set up ChipS with a permission set for upgrades. This should be done to ChipS and that part of M designated by P_0 set to ReadOnly *before* ChipS is programmed with K_U. If K_S is programmed with K_U first, there is a risk of someone obtaining a half-setup ChipS and changing all of M_U instead of only the sections specified by Q.

10   In addition, CountRemaining in ChipS needs to be setup (including making it ReadOnly in P_S) before ChipS is programmed with K_U. ChipS should therefore be programmed to only perform a limited number of SignM operations (thereby limiting compromise exposure if a ChipS is stolen). Thus ChipS would itself need to be upgraded with a new CountRemaining every so often.

15   ### 9.4.3   Updating permissions for future writes
In order to reduce exposure to accidental and malicious attacks on P (and certain parts of M), only authorized users are allowed to update P. Writes to P are the same as authorized writes to M, except that they update P_n instead of M. Initially (at manufacture), P is set to be Read/Write for all parts of M. As different processes fill up different parts of M, they can be sealed against future
20   change by updating the permissions. Updating a chip's P_0 changes permissions for unauthorized writes, and updating P_{n+1} changes permissions for authorized writes with key K_n.

P_n is only allowed to change to be a more restrictive form of itself. For example, initially all parts of M have permissions of Read/Write. A permission of Read/Write can be updated to Decrement Only
25   or Read Only. A permission of Decrement Only can be updated to become Read Only. A Read Only permission cannot be further restricted.
In this transaction protocol, the System's chip is referred to as ChipS, and the chip being updated is referred to as ChipU. Each chip distrusts the other.

30   The protocol requires the following publicly available functions in ChipU:
Random[]   Returns R (does not advance R).
SetPermission[n,p,X,Y,Z] Advances R, and updates P_p according to Y and returns 1 followed by the resultant P_p only if $S_{Kn}[R|X|Y|C_2]$ = Z. Otherwise returns 0. P_p can only become more restricted. Passing in 0 for any permission leaves it unchanged (passing in
35   Y=0 returns the current P_p).

Authenticated writes of permissions require that the System has access to a ChipS that is capable of generating appropriate signatures. ChipS requires the following variables and function:

CountRemaining   Part of M that contains the number of signatures that ChipS is allowed to generate. Decrements with each successful call to SignM and SignP. Permissions in ChipS's P_0 for this part of M needs to be ReadOnly once ChipS has been setup. Therefore CountRemaining can only be updated by another ChipS that will perform updates to that part of M (assuming ChipS's P_n allows that part of M to be updated).

SignP[n,X,Y]   Advances R, decrements CountRemaining and returns R and $S_{Kn}[X|R|Y|C_2]$ only if CountRemaining > 0. Otherwise returns all 0s. The time taken to calculate and compare signatures must be independent of data content.

To update ChipU's P_n:

a. System calls ChipU's Random function;

b. ChipU returns R_U to System;

c. System calls ChipS's SignP function, passing in n1, R_U and P_D (the desired P to be written to ChipU);

d. ChipS produces R_S and $S_{Kn1}[R_U|R_S|P_D|C_2]$ if it is still permitted to produce signatures.

e. If values returned in d are non zero, then System can then call ChipU's SetPermission function with n2, the desired permission entry p, R_S, P_D and $S_{Kn1}[R_U|R_S|P_D|C_2]$.

f. ChipU verifies the received signature against $S_{Kn2}[R_U|R_S|P_D|C_2]$ and applies P_D to P_n if the signature matches

g. System checks 1st output parameter. 1 = success, 0 = failure.

The choice of n1 and n2 must be such that ChipU's K_{n1} = ChipS's K_{n2}.

The data flow for authenticated writes to permissions is shown in Figure 340 below.

9.4.4 Protecting M in a multiple key system

To protect the appropriate part of M, the SetPermission function must be called *after* the part of M has been set to the desired value.

For example, if adding a serial number to an area of M that is currently ReadWrite so that noone is permitted to update the number again:

•   the Write function is called to write the serial number to M

•   SetPermission is called for n = {1, ..., N} to set that part of M to be ReadOnly for authorized writes using key n-1.

676

- SetPermission is called for 0 to set that part of M to be ReadOnly for non-authorized writes

For example, adding a consumable value to M such that only keys 1-2 can update it, and keys 0, and 3-N cannot:

5
- the Write function is called to write the amount of consumable to M
- SetPermission is called for n = {1, 4, 5, ..., N-1} to set that part of M to be ReadOnly for authorized writes using key n-1. This leaves keys 1 and 2 with ReadWrite permissions.
- SetPermission is called for 0 to set that part of M to be DecrementOnly for non-authorized writes. This allows the amount of consumable to decrement.

10

It is possible for someone who knows a key to further restrict other keys, but it is not in anyone's interest to do so.

9.5   PROGRAMMING K

15   In this case, we have a factory chip (*ChipF*) connected to a System. The System wants to program the key in another chip (*ChipP*). System wants to avoid passing the new key to ChipP in the clear, and also wants to avoid the possibility of the key-upgrade message being replayed on another ChipP (even if the user doesn't know the key).

20   The protocol is a simple extension of the single key protocol in that it assumes that ChipF and ChipP already share a secret key K_{old}. This key is used to ensure that only a chip that knows K_{old} can set K_{new}.

The protocol requires the following publicly available functions in ChipP:

25   Random[]   Returns R (does not advance R).

ReplaceKey[n, X, Y, Z]   Replaces K_n by $S_{Kn}[R|X|C_3] \oplus Y$, advances R, and returns 1 only if $S_{Kn}[X|Y|C_3] = Z$. Otherwise returns 0. The time taken to calculate signatures and compare values must be identical for all inputs.

30   And the following data and functions in ChipF:

CountRemaining   Part of M with contains the number of signatures that ChipF is allowed to generate. Decrements with each successful call to GetProgramKey. Permissions in P for this part of M needs to be ReadOnly once ChipF has been setup. Therefore can only be updated by a ChipS that has

35   authority to perform updates to that part of M.

K_{new} The new key to be transferred from ChipF to ChipP. Must not be visible.

SetPartialKey[X,Y]	If word X of K_{new} has not yet been set, set word X of K_{new} to Y and return 1. Otherwise return 0. This function allows K_{new} to be programmed in multiple steps, thereby allowing different people or systems to know different parts of the key (but not the whole K_{new}). K_{new} is stored in ChipF's flash memory. Since there is a small number of ChipFs, it is theoretically not necessary to store the inverse of K_{new}, but it is stronger protection to do so.
GetProgramKey[n, X]	Advances R_F, decrements CountRemaining, outputs R_F, the encrypted key $S_{Kn}[X|R_F|C_3]\oplus K_{new}$ and a signature of the first two outputs plus C_3 if CountRemaining>0. Otherwise outputs 0. The time to calculate the encrypted key & signature must be identical for all inputs.

To update P's key :

a. System calls ChipP's Random function;

b. ChipP returns R_P to System;

c. System calls ChipF's GetProgramKey function, passing in n1 (the desired key to use) and the result from b;

d. ChipF updates R_F, then calculates and returns R_F, $S_{Kn1}[R_P|R_F|C_3]\oplus K_{new}$, and $S_{Kn1}[R_F|S_{Kn1}[R_P|R_F|C_3]\oplus K_{new}|C_3]$;

e. If the response from d is not 0, System calls ChipP's ReplaceKey function, passing in n2 (the key to use in ChipP) and the response from d;

f. System checks response from ChipP. If the response is 1, then K_{Pn2} has been correctly updated to K_{new}. If the response is 0, K_{Pn2} has not been updated.

The choice of n1 and n2 must be such that ChipF's K_{n1} = ChipP's K_{n2}.

The data flow for key updates is shown in Figure 341 below.

Note that K_{new} is never passed in the open. An attacker could send its own R_P, but cannot produce $S_{Kn1}[R_P|R_F|C_3]$ without K_{n1}. The signature based on K_{new} is sent to ensure that ChipP will be able to determine if either of the first two parameters have been changed en route.

CountRemaining needs to be setup in M_F (including making it ReadOnly in P) before ChipF is programmed with K_P. ChipF should therefore be programmed to only perform a limited number of GetProgramKey operations (thereby limiting compromise exposure if a ChipF is stolen). An authorized ChipS can be used to update this counter if neccesary (see Section 9.4 on page 673).

9.5.1 Chicken and Egg

As with the single key protocol, for the Program Key protocol to work, both ChipF and ChipP must both know K_{old}. Obviously both chips had to be programmed with K_{old}, and thus K_{old} can be thought of as an older K_{new}: K_{old} can be placed in chips if another ChipF knows K_{older}, and so on.

5 Although this process allows a chain of reprogramming of keys, with each stage secure, at some stage the very first key (K_{first}) must be placed in the chips. K_{first} is in fact programmed with the chip's microcode at the manufacturing test station as the last step in manufacturing test. K_{first} can be a manufacturing batch key, changed for each batch or for each customer etc, and can have as short a life as desired. Compromising K_{first} need not result in a complete compromise of the chain of Ks.

10

Depending on the reprogramming requirements, K_{first} can be the same or different for all K_n.

10   Multiple Keys Multiple Memory Vectors
10.1   PROTOCOL BACKGROUND
15 This protocol set is a slight restriction of the multiple key single memory vector protocol set, and is the expected protocol. It is a restriction in that M has been optimized for Flash memory utilization.

M is broken into multiple memory vectors (semi-fixed and variable components) for the purposes of optimizing flash memory utilization. Typically M contains some parts that are fixed at some stage of 20 the manufacturing process (eg a batch number, serial number etc), and once set, are not ever updated. This information does not contain the amount of consumable remaining, and therefore is not read or written to with any great frequency.

We therefore define M_0 to be the M that contains the frequently updated sections, and the 25 remaining Ms to be rarely written to. Authenticated writes only write to M_0, and non-authenticated writes can be directed to a specific M_n. This reduces the size of permissions that are stored in the QA Chip (since key-based writes are not required for Ms other than M_0). It also means that M_0 and the remaining Ms can be manipulated in different ways, thereby increasing flash memory longevity.

30 10.2   REQUIREMENTS OF PROTOCOL
Each QA Chip contains the following values:
N   The maximum number of keys known to the chip.
T   The number of vectors M is broken into.
K_N Array of N secret keys used for calculating $F_{Kn}[X]$ where K_n is the nth element of the array.
35 Each K_n must not be stored directly in the QA Chip . Instead, each chip needs to store a single random number R_K (different for each chip), $K_n \oplus R_K$, and $\neg K_n \oplus R_K$. The stored $K_n \oplus R_K$ can be

XORed with R_K to obtain the real K_n. Although $\neg K_n \oplus R_K$ must be stored to protect against differential attacks, it is not used.

R   Current random number used to ensure time varying messages. Each chip instance must be seeded with a different initial value. Changes for each signature generation.

M_T Array of T memory vectors. Only M_0 can be written to with an authorized write, while all Ms can be written to in an unauthorized write. Writes to M_0 are optimized for Flash usage, while updates to any other M_n are expensive with regards to Flash utilization, and are expected to be only performed once per section of M_n. M_1 contains T and N in ReadOnly form so users of the chip can know these two values.

P_{T+N} T+N element array of access permissions for each part of M. Entries n={0... T-1} hold access permissions for non-authenticated writes to M_n (no key required). Entries n={T to T+N-1}hold access permissions for authenticated writes to M_0 for K_n. Permission choices for each part of M are Read Only, Read/Write, and Decrement Only.

C   3 constants used for generating signatures. C_1, C_2, and C_3 are constants that pad out a submessage to a hashing boundary, and all 3 must be different.

Each QA Chip contains the following private function:

$S_{Kn}[N,X]$ *Internal function only.* Returns $S_{Kn}[X]$, the result of applying a digital signature function S to X based upon the appropriate key K_n. The digital signature must be long enough to counter the chances of someone generating a random signature. The length depends on the signature scheme chosen, although the scheme chosen for the QA Chip is HMAC-SHA1, and therefore the length of the signature is 160 bits.

Additional functions are required in certain QA Chips, but these are described as required.

10.3   READS

As with the previous scenarios, we have a trusted chip (*ChipT*) connected to a System. The System wants to authenticate an object that contains a non-trusted chip (*ChipA*). In effect, the System wants to know that it can securely read a memory vector (M_t) from ChipA: to be sure that ChipA is valid and that M has not been altered.

The protocol requires the following publicly available functions:

Random[]   Returns R (does not advance R).

Read[n, t, X]   Advances R, and returns R, M_t, $S_{Kn}[X|R|C_1|M_t]$. The time taken to calculate the signature must not be based on the contents of X, R, M_t, or K. If t is invalid, the function assumes t=0.

Test[n,X, Y, Z]   Advances R and returns 1 if $S_{Kn}[R|X|C_1|Y] = Z$. Otherwise returns 0. The time taken to calculate and compare signatures must be independent of data content.

5   To authenticate ChipA and read ChipA's memory M:

a.   System calls ChipT's Random function;

b.   ChipT returns R_T to System;

c.   System calls ChipA's Read function, passing in some key number n1, the desired M number t, and the result from b;

10   d.   ChipA updates R_A, then calculates and returns R_A, M_{At}, $S_{KAn1}[R_T|R_A|C_1|M_{At}]$;

e.   System calls ChipT's Test function, passing in n2, R_A, M_{At}, $S_{KAn1}[R_T|R_A|C_1|M_{At}]$;

f.   System checks response from ChipT. If the response is 1, then ChipA is considered authentic. If 0, ChipA is considered invalid.

15   The choice of n1 and n2 must be such that ChipA's K_{n1} = ChipT's K_{n2}.

The data flow for read authentication is shown in Figure 342 below.

The protocol allows System to simply pass data from one chip to another, with no special
20   processing. The protection relies on ChipT being trusted, even though System does not know K.

When ChipT is physically separate from System (eg is chip on a board connected to System) System *must also occassionally* (based on system clock for example) call ChipT's Test function with bad data, expecting a 0 response. This is to prevent someone from inserting a fake ChipT into
25   the system that always returns 1 for the Test function.

It is important that n1 is chosen by System. Otherwise ChipA would need to return N_A sets of signatures for each read, since ChipA does not know which of the keys will satisfy ChipT. Similarly, system must also choose n2, so it can potentially restrict the number of keys in ChipT that are
30   matched against (otherwise ChipT would have to match against all its keys). This is important in order to restrict how different keys are used. For example, say that ChipT contains 6 keys, keys 0-2 are for various printer-related upgrades, and keys 3-6 are for inks. ChipA contains say 4 keys, one key for each printer model. At power-up, System goes through each of chipA's keys 0-3, trying each out against ChipT's keys 3-6. System doesn't try to match against ChipT's keys 0-2. Otherwise
35   knowledge of a speed-upgrade key could be used to provide ink QA Chip chips. This matching needs to be done only once (eg at power up). Once matching keys are found, System can continue to use those key numbers.

Since System needs to know N_T, N_A, and T_A, part of M_1 is used to hold N (eg in Read Only form), and the system can obtain it by calling the Read function, passing in key 0 and t=1.

10.4   WRITES

As with the previous scenarios, the System wants to update M_t in ChipU. As before, this can be done in a non-authenticated and authenticated way.

10.4.1   Non-authenticated writes

This is the most frequent type of write, and takes place between the System / consumable during normal everyday operation for M_0, and during the manufacturing process for M_t.

In this kind of write, System wants to change M subject to P. For example, the System could be decrementing the amount of consumable remaining. Although *System does not need to know and of the Ks or even have access to a trusted chip* to perform the write, System must follow a non-authenticated write by an authenticated read if it needs to know that the write was successful.

The protocol requires the following publicly available function:

Write[t, X]   Writes X over those parts of M_t subject to P_t and the existing value for M.

To authenticate a write of M_{new} to ChipA's memory M:

a. System calls ChipU's Write function, passing in M_{new};
b. The authentication procedure for a Read is carried out (see Section 9.3 on page 671);
c. If ChipU is authentic and M_{new} = M returned in b, the write succeeded. If not, it failed.

10.4.2   Authenticated writes

In the multiple memory vectors protocol, only M_0 can be written to an an authenticated way. This is because only M_0 is considered to have components that need to be upgraded.

In this kind of write, System wants to change Chip U's M_0 in an authorized way, without being subject to the permissions that apply during normal operation. For example, the consumable may be at a refilling station and the normally Decrement Only section of M_0 should be updated to include the new valid consumable. In this case, the chip whose M_0 is being updated must authenticate the writes being generated by the external System and in addition, apply the appropriate permission for the key to ensure that only the correct parts of M_0 are updated. Having a different permission for each key is required as when multiple keys are involved, all keys should not necessarily be given open access to M_0. For example, suppose M_0 contains printer speed and a counter of money

available for franking. A ChipS that updates printer speed should not be capable of updating the amount of money. Since $P_{0...T-1}$ is used for non-authenticated writes, each K_n has a corresponding permission P_{T+n} that determines what can be updated in an authenticated write.

5   In this transaction protocol, the System's chip is referred to as ChipS, and the chip being updated is referred to as ChipU. Each chip distrusts the other.

The protocol requires the following publicly available functions in ChipU:

Read[n, t, X]   Advances R, and returns R, M_t, $S_{Kn}[X|R|C_1|M_t]$. The time taken to cal-
10   culate the signature must not be based on the contents of X, R, M_t, or K.

WriteA[n, X, Y, Z]   Advances R, replaces M_0 by Y subject to P_{T+n}, and returns 1 only if $S_{Kn}[R|X|C_1|Y] = Z$. Otherwise returns 0. The time taken to calculate and compare signatures must be independent of data content. This
15   function is identical to ChipT's Test function except that it additionally writes Y subject to P_{T+n} to its M when the signature matches.

Authenticated writes require that the System has access to a ChipS that is capable of generating appropriate signatures. ChipS requires the following variables and function:

20   CountRemaining   Part of M that contains the number of signatures that ChipS is allowed to generate. Decrements with each successful call to SignM and SignP. Permissions in ChipS's $P_{0..T-1}$ for this part of M needs to be ReadOnly once ChipS has been setup. Therefore CountRemaining can only be updated by another ChipS that will
25   perform updates to that part of M (assuming ChipS's P allows that part of M to be updated).

Q   Part of M that contains the write permissions for updating ChipU's M. By adding Q to ChipS we allow different ChipSs that can update different parts of M_U. Permissions in ChipS's $P_{0..T-1}$ for this part of M
30   needs to be ReadOnly once ChipS has been setup. Therefore Q can only be updated by another ChipS that will perform updates to that part of M.

SignM[n,V,W,X,Y,Z]   Advances R, decrements CountRemaining and returns R, Z_{QX} (Z applied to X with permissions Q), $S_{Kn}[W|R|C_1|Z_{QX}]$ only if Y =
35                     $S_{Kn}[V|W|C_1|X]$ and CountRemaining > 0. Otherwise returns all 0s. The time taken to calculate and compare signatures must be independent of data content.

683

To update ChipU's M vector:

a. System calls ChipU's Read function, passing in n1, 0 and 0 as the input parameters;

b. ChipU produces R_U, M_{U0}, $S_{Kn1}[0|R_U|C_1|M_{U0}]$ and returns these to System;

c. System calls ChipS's SignM function, passing in n2 (the key to be used in ChipS), 0 (as used in a), R_U, M_{U0}, $S_{Kn1}[0|R_U|C_1|M_{U0}]$, and M_D (the desired vector to be written to ChipU);

d. ChipS produces R_S, M_{QD} (processed by running M_D against M_{U0} using Q) and $S_{Kn2}[R_U|R_S|C_1|M_{QD}]$ if the inputs were valid, and 0 for all outputs if the inputs were not valid.

e. If values returned in d are non zero, then ChipU is considered authentic. System can then call ChipU's WriteA function with these values from d.

f. ChipU should return a 1 to indicate success. A 0 should only be returned if the data generated by ChipS is incorrect (e.g. a transmission error).

The choice of n1 and n2 must be such that ChipU's K_{n1} = ChipS's K_{n2}.

The data flow for authenticated writes is shown in Figure 343 below.

Note that Q in ChipS is part of ChipS's M. This allows a user to set up ChipS with a permission set for upgrades. This should be done to ChipS and that part of M designated by $P_{0..T-1}$ set to ReadOnly *before* ChipS is programmed with K_U. If K_S is programmed with K_U first, there is a risk of someone obtaining a half-setup ChipS and changing all of M_U instead of only the sections specified by Q.

In addition, CountRemaining in ChipS needs to be setup (including making it ReadOnly in P_S) before ChipS is programmed with K_U. ChipS should therefore be programmed to only perform a limited number of SignM operations (thereby limiting compromise exposure if a ChipS is stolen). Thus ChipS would itself need to be upgraded with a new CountRemaining every so often.

10.4.3 Updating permissions for future writes

In order to reduce exposure to accidental and malicious attacks on P (and certain parts of M), only authorized users are allowed to update P. Writes to P are the same as authorized writes to M, except that they update P_n instead of M. Initially (at manufacture), P is set to be Read/Write for all M. As different processes fill up different parts of M, they can be sealed against future change by updating the permissions. Updating a chip's $P_{0..T-1}$ changes permissions for unauthorized writes to M_n, and updating $P_{T..T+N-1}$ changes permissions for authorized writes with key K_n.

P_n is only allowed to change to be a more restrictive form of itself. For example, initially all parts of M have permissions of Read/Write. A permission of Read/Write can be updated to Decrement Only

or Read Only. A permission of Decrement Only can be updated to become Read Only. A Read Only permission cannot be further restricted.

In this transaction protocol, the System's chip is referred to as ChipS, and the chip being updated is referred to as ChipU. Each chip distrusts the other.

The protocol requires the following publicly available functions in ChipU:

Random[]   Returns R (does not advance R).

SetPermission[n,p,X,Y,Z] Advances R, and updates P_p according to Y and returns 1 followed by the resultant P_p only if $S_{Kn}[R|X|Y|C_2] = Z$. Otherwise returns 0. P_p can only become more restricted. Passing in 0 for any permission leaves it unchanged (passing in Y=0 returns the current P_p).

Authenticated writes of permissions require that the System has access to a ChipS that is capable of generating appropriate signatures. ChipS requires the following variables and function:

CountRemaining	Part of ChipS's M_0 that contains the number of signatures that ChipS is allowed to generate. Decrements with each successful call to SignM and SignP. Permissions in ChipS's $P_{0..T-1}$ for this part of M_0 needs to be ReadOnly once ChipS has been setup. Therefore CountRemaining can only be updated by another ChipS that will perform updates to that part of M_0 (assuming ChipS's P_n allows that part of M_0 to be updated).
SignP[n,X,Y]	Advances R, decrements CountRemaining and returns R and $S_{Kn}[X|R|Y|C_2]$ only if CountRemaining > 0. Otherwise returns all 0s. The time taken to calculate and compare signatures must be independent of data content.

To update ChipU's P_n:

a. System calls ChipU's Random function;

b. ChipU returns R_U to System;

c. System calls ChipS's SignP function, passing in n1, R_U and P_D (the desired P to be written to ChipU);

d. ChipS produces R_S and $S_{Kn1}[R_U|R_S|P_D|C_2]$ if it is still permitted to produce signatures.

e. If values returned in d are non zero, then System can then call ChipU's SetPermission function with n2, the desired permission entry p, R_S, P_D and $S_{Kn1}[R_U|R_S|P_D|C_2]$.

f. ChipU verifies the received signature against $S_{Kn2}[R_U|R_S|P_D|C_2]$ and applies P_D to P_n if the signature matches

g. System checks 1st output parameter. 1 = success, 0 = failure.

The choice of n1 and n2 must be such that ChipU's K_{n1} = ChipS's K_{n2}.

5   The data flow for authenticated writes to permissions is shown in Figure 344 below.

10.4.4   Protecting M in a multiple key multiple M system

To protect the appropriate part of M_n against unauthorized writes, call SetPermissions[n] for n = 0 to

T-1. To protect the appropriate part of M_0 against authorized writes with key n, call

10   SetPermissions[T+n] for n=0 to N-1.

Note that only M_0 can be written in an authenticated fashion.

Note that the SetPermission function must be called *after* the part of M has been set to the desired

15   value.

For example, if adding a serial number to an area of M_1 that is currently ReadWrite so that noone is
permitted to update the number again:
*   the Write function is called to write the serial number to M_1

20   *   SetPermission(1) is called for to set that part of M to be ReadOnly for non-authorized writes.

If adding a consumable value to M_0 such that only keys 1-2 can update it, and keys 0, and 3-N
cannot:
*   the Write function is called to write the amount of consumable to M

25   *   SetPermission is called for 0 to set that part of M_0 to be DecrementOnly for *non-authorized*
     writes. This allows the amount of consumable to decrement.
*   SetPermission is called for n = {T, T+3, T+4 ..., T+N-1} to set that part of M_0 to be ReadOnly
     for *authorized* writes using all but keys 1 and 2. This leaves keys 1 and 2 with ReadWrite
     permissions to M_0.

30

It is possible for someone who knows a key to further restrict other keys, but it is not in anyone's
interest to do so.

10.5   PROGRAMMING K

35   This section is identical to the multiple key single memory vector (Section 9.5 on page 677). It is
repeated here with mention to M_0 instead of M for CountRemaining.

In this case, we have a factory chip (*ChipF*) connected to a System. The System wants to program the key in another chip (*ChipP*). System wants to avoid passing the new key to ChipP in the clear, and also wants to avoid the possibility of the key-upgrade message being replayed on another ChipP (even if the user doesn't know the key).

5

The protocol is a simple extension of the single key protocol in that it assumes that ChipF and ChipP already share a secret key K_{old}. This key is used to ensure that only a chip that knows K_{old} can set K_{new}.

10   The protocol requires the following publicly available functions in ChipP:

Random[]   Returns R (does not advance R).

ReplaceKey[n, X, Y, Z]   Replaces K_n by $S_{Kn}[R|X|C_3] \oplus Y$, advances R, and returns 1 only if $S_{Kn}[X|Y|C_3] = Z$. Otherwise returns 0. The time taken to calculate signatures and compare values must be identical for all inputs.

15   And the following data and functions in ChipF:

CountRemaining	Part of M_0 with contains the number of signatures that ChipF is allowed to generate. Decrements with each successful call to GetProgramKey. Permissions in P for this part of M_0 needs to be ReadOnly once ChipF has been setup. Therefore can only be updated by a ChipS that has authority to perform updates to that part of M_0.
K_{new}	The new key to be transferred from ChipF to ChipP. Must not be visible.
SetPartialKey[X,Y]	If word X of K_{new} has not yet been set, set word X of K_{new} to Y and return 1. Otherwise return 0. This function allows K_{new} to be programmed in multiple steps, thereby allowing different people or systems to know different parts of the key (but not the whole K_{new}). K_{new} is stored in ChipF's flash memory. Since there is a small number of ChipFs, it is theoretically not necessary to store the inverse of K_{new}, but it is stronger protection to do so.
GetProgramKey[n, X]	Advances R_F, decrements CountRemaining, outputs R_F, the encrypted key $S_{Kn}[X|R_F|C_3] \oplus K_{new}$ and a signature of the first two outputs plus C_3 if CountRemaining>0. Otherwise outputs 0. The time to calculate the encrypted key & signature must be identical for all inputs.

35

To update P's key :

a. System calls ChipP's Random function;

b. ChipP returns R_P to System;

c. System calls ChipF's GetProgramKey function, passing in n1 (the desired key to use) and the result from b;

5   d. ChipF updates R_F, then calculates and returns R_F, $S_{Kn1}[R_P|R_F|C_3]\oplus K_{new}$, and $S_{Kn1}[R_F|S_{Kn1}[R_P|R_F|C_3]\oplus K_{new}|C_3]$;

e. If the response from d is not 0, System calls ChipP's ReplaceKey function, passing in n2 (the key to use in ChipP) and the response from d;

f. System checks response from ChipP. If the response is 1, then K_{Pn2} has been correctly updated

10   to K_{new}. If the response is 0, K_{Pn2} has not been updated.

The choice of n1 and n2 must be such that ChipF's K_{n1} = ChipP's K_{n2}.

The data flow for key updates is shown in Figure 345below.

Note that K_{new} is never passed in the open. An attacker could send its own R_P, but cannot produce $S_{Kn1}[R_P|R_F|C_3]$ without K_{n1}. The signature based on K_{new} is sent to ensure that ChipP will be able to

15   determine if either of the first two parameters have been changed en route.

CountRemaining needs to be setup in M_{F0} (including making it ReadOnly in P) before ChipF is programmed with K_P. ChipF should therefore be programmed to only perform a limited number of GetProgramKey operations (thereby limiting compromise exposure if a ChipF is stolen). An authorized ChipS can be used to update this counter if neccesary (see Section 9.4 on page 673).

20

10.5.1   Chicken and Egg

As with the single key protocol, for the Program Key protocol to work, both ChipF and ChipP must both know K_{old}. Obviously both chips had to be programmed with K_{old}, and thus K_{old} can be thought of as an older K_{new}: K_{old} can be placed in chips if another ChipF knows K_{older}, and so on.

25

Although this process allows a chain of reprogramming of keys, with each stage secure, at some stage the very first key (K_{first}) must be placed in the chips. K_{first} is in fact programmed with the chip's microcode at the manufacturing test station as the last step in manufacturing test. K_{first} can be a manufacturing batch key, changed for each batch or for each customer etc, and can have as short

30   a life as desired. Compromising K_{first} need not result in a complete compromise of the chain of Ks. Depending on reprogramming requirements, K_{first} can be the same or different for all K_n.

10.5.2   Security Note

Different ChipFs should have different R_F values to prevent K_{new} from being determined as follows:

35   The attacker needs 2 ChipFs, both with the same R_F and K_n but different values for K_{new}. By knowing K_{new1} the attacker can determine K_{new2}. The size of R_F is 2^{160}, and assuming a lifespan of approximately 2^{32} Rs, an attacker needs about 2^{60} ChipFs with the same K_n to locate the correct

chip. Given that there are likely to be only hundreds of ChipFs with the same K_n, this is not a likely attack. The attack can be eliminated completely by making C_3 different per chip and transmitting it with the new signature.

11 Summary of functions for all protocols

All protocol sets, whether single key, multiple key, single M or multiple M, all rely on the same set of functions. The function set is listed here:

11.1 ALL CHIPS

Since every chip must act as ChipP, ChipA and potentially ChipU, *all* chips require the following functions:

- Random
- ReplaceKey
- Read
- Write
- WriteA
- SetPermissions

11.2 CHIPT

Chips that are to be used as ChipT also require:

- Test

11.3 CHIPS

Chips that are to be used as ChipS also require either or both of:

- SignM
- SignP

11.4 CHIPF

Chips that are to be used as ChipF also require:

- SetPartialKey
- GetProgramKey

12 Remote Upgrades

12.1 BASIC REMOTE UPGRADES

Regardless of the number of keys and the number of memory vectors, the use of authenticated reads and writes, and of replacing a new key without revealing K_{new} or K_{old} allows the possibility of

remote upgrades of ChipU and ChipP. The upgrade typically involves a remote server and follows two basic steps:

a.   During the first stage of the upgrade, the remote system authenticates the user's system to ensure the user's system has the setup that it claims to have.

5   b.   During the second stage of the upgrade, the user's system authenticates the remote system to ensure that the upgrade is from a trusted source.

12.1.1   User requests upgrade

The user requests that he wants to upgrade. This can be done by running a specific upgrade
10   application on the user's computer, or by visiting a specific website.

12.1.2   Remote system gathers info securely about user's current setup

In this step, the remote system determines the current setup for the user. The current setup must be authenticated, to ensure that the user truly has the setup that is claimed. Traditionally, this has
15   been by checking the existence of files, generating checksums from those files, or by getting a serial number from a hardware dongle, although these traditional methods have difficulties since they can be generated locally by "hacked" software.

The authenticated read protocol described in Section 8.3 on page 664 can be used to accomplish
20   this step. The use of random numbers has the advantage that the local user cannot capture a successful transaction and play it back on another computer system to fool the remote system.

12.1.3   Remote system gives user choice of upgrade possibilities & user chooses

If there is more than one upgrade possibility, the various upgrade options are now presented to the
25   user. The upgrade options could vary based on a number of factors, including, but not limited to:

•   current user setup

•   user's preference for payment schemes (e.g. single payment vs. multiple payment)

•   number of other products owned by user

30   The user selects an appropriate upgrade and pays if necessary (by some scheme such as via a secure web site). What is important to note here is that the user chooses a specific upgrade and commences the upgrade operation.

12.1.4   Remote system sends upgrade request to local system

35   The remote system now instructs the local system to perform the upgrade. However, the local system can only accept an upgrade from the remote system if the remote system is also

authenticated. This is effectively an authenticated write. The use of R_U in the signature prevents the upgrade message from being replayed on another ChipU.

If multiple keys are used, and each chip has a unique key, the remote system can use a serial number obtained from the current setup (authenticated by a common key) to lookup the unique key for use in the upgrade. Although the random number provides time varying messages, use of an unknown K that is different for each chip means that collection and examination of messages and their signatures is made even more difficult.

12.2 OEM UPGRADES

OEM upgrades are effectively the same as remote upgrades, except that the user interacts with an OEM server for upgrade selection. The OEM server may send sub-requests to the manufacturer's remote server to provide authentication, upgrade availability lists, and base-level pricing information.

An additional level of authentication may be incorporated into the protocol to ensure that upgrade requests are coming from the OEM server, and not from a 3rd party. This can readily be incorporated into both authentication steps.

13 Choice of Signature Function

Given that all protocols make use of keyed signature functions, the choice of function is examined here.

Table 232 outlines the attributes of the applicable choices (see Section 5.2 on page 629 and Section 5.5 on page 636 for more information). The attributes are phrased so that the attribute is seen as an advantage.

Table 232. Attributes of Applicable Signature Functions

	Triple DES	Blowfish	RC5	IDEA	Random Sequences	HMAC-MD5	HMAC-SHA1	HMAC-RIPEMD160
Free of patents	●	●			●	●	●	●
Random key generation						●	●	●
Can be exported from the USA					●	●	●	●

Fast		●				●	●	●
Preferred Key Size (bits) for use in this application	168[1]	128	128	128	512	128	160	160
Block size (bits)	64	64	64	64	256	512	512	512
Cryptanalysis Attack-Free (apart from weak keys)	●	●			●		●	●
Output size given input size N	≥N	≥N	≥N	≥N	128	128	160	160
Low storage requirements					●	●	●	●
Low silicon complexity					●	●	●	●
NSA designed	●						●	

An examination of Table 232 shows that the choice is effectively between the 3 HMAC constructs and the Random Sequence. The problem of key size and key generation eliminates the Random Sequence. Given that a number of attacks have already been carried out on MD5 and since the hash result is only 128 bits, HMAC-MD5 is also eliminated. The choice is therefore between HMAC-SHA1 and HMAC-RIPEMD160. Of these, SHA-1 is the preferred function, since:

- SHA-1 has been more extensively cryptanalyzed without being broken;
- SHA-1 requires slightly less intermediate storage than RIPE-MD-160;
- SHA-1 is algorithmically less complex than RIPE-MD-160;

Although SHA-1 is slightly faster than RIPE-MD-160, this was not a reason for choosing SHA-1.

13.1   HMAC-SHA1

The mechanism for authentication is the HMAC-SHA1 algorithm. This section examines the HMAC-SHA1 algorithm in greater detail than covered so far, and describes an optimization of the algorithm that requires fewer memory resources than the original definition.

13.1.1   HMAC

Given the following definitions:

- H   = the hash function (e.g. MD5 or SHA-1)
- n   = number of bits output from H (e.g. 160 for SHA-1, 128 bits for MD5)
- M   = the data to which the MAC function is to be applied
- K   = the secret key shared by the two parties
- ipad   = 0x36 repeated 64 times

---

[1]   Only gives protection equivalent to 112-bit DES

- opad = 0x5C repeated 64 times

The HMAC algorithm is as follows:

a. Extend K to 64 bytes by appending 0x00 bytes to the end of K

b. XOR the 64 byte string created in (1) with ipad

c. append data stream M to the 64 byte string created in (2)

d. Apply H to the stream generated in (3)

e. XOR the 64 byte string created in (1) with opad

f. Append the H result from (4) to the 64 byte string resulting from (5)

g. Apply H to the output of (6) and output the result

Thus:

$$HMAC[M] = H[(K \oplus opad) \mid H[(K \oplus ipad) \mid M]]$$

The HMAC-SHA1 algorithm is simply HMAC with H = SHA-1.

13.1.2 SHA-1

The SHA1 hashing algorithm is described in the context of other hashing algorithms in Section 5.5.3.3 on page 640, and completely defined in [28]. The algorithm is summarized here.

Nine 32-bit constants are defined in Table 233. There are 5 constants used to initialize the chaining variables, and there are 4 additive constants.

Table 233. Constants used in SHA-1

Initial Chaining Values		Additive Constants	
h_1	0x67452301	y_1	0x5A827999
h_2	0xEFCDAB89	y_2	0x6ED9EBA1
h_3	0x98BADCFE	y_3	0x8F1BBCDC
h_4	0x10325476	y_4	0xCA62C1D6
h_5	0xC3D2E1F0		

Non-optimized SHA-1 requires a total of 2912 bits of data storage:

- Five 32-bit chaining variables are defined: H_1, H_2, H_3, H_4 and H_5.

- Five 32-bit working variables are defined: A, B, C, D, and E.

- One 32-bit temporary variable is defined: t.

- Eighty 32-bit temporary registers are defined: X_{0-79}.

The following functions are defined for SHA-1:

Table 234. Functions used in SHA-1

Symbolic Nomenclature	Description
+	Addition modulo 2^{32}
$X \ll Y$	Result of rotating X left through Y bit positions
$f(X, Y, Z)$	$(X \wedge Y) \vee (\neg X \wedge Z)$
$g(X, Y, Z)$	$(X \wedge Y) \vee (X \wedge Z) \vee (Y \wedge Z)$
$h(X, Y, Z)$	$X \oplus Y \oplus Z$

5   The hashing algorithm consists of firstly padding the input message to be a multiple of 512 bits and initializing the chaining variables $H_{1\text{-}5}$ with $h_{1\text{-}5}$. The padded message is then processed in 512-bit chunks, with the output hash value being the final 160-bit value given by the concatenation of the chaining variables: $H_1 \mid H_2 \mid H_3 \mid H_4 \mid H_5$.

10   The steps of the SHA-1 algorithm are now examined in greater detail.

13.1.2.1 Step 1. Preprocessing

The first step of SHA-1 is to pad the input message to be a multiple of 512 bits as follows and to initialize the chaining variables.

15

Table 235. Steps to follow to preprocess the input message

Pad the input message	Append a 1 bit to the message
	Append 0 bits such that the length of the padded message is 64-bits short of a multiple of 512 bits.
	Append a 64-bit value containing the length in bits of the original input message. Store the length as most significant bit through to least significant bit.
Initialize the chaining variables	$H_1 \leftarrow h_1, H_2 \leftarrow h_2, H_3 \leftarrow h_3, H_4 \leftarrow h_4, H_5 \leftarrow h_5$

13.1.2.2 Step 2. Processing

The padded input message is processed in 512-bit blocks. Each 512-bit block is in the form of 16 × 32-bit words, referred to as $InputWord_{0-15}$.

5   Table 236. Steps to follow for each 512 bit block ($InputWord_{0-15}$)

Copy the 512 input bits into X_{0-15}	For j=0 to 15 $X_j = InputWord_j$
Expand X_{0-15} into X_{16-79}	For j=16 to 79 $Xj \leftarrow ((X_{j-3} \oplus X_{j-8} \oplus X_{j-14} \oplus X_{j-16}) \ll 1)$
Initialize working variables	$A \leftarrow H_1, B \leftarrow H_2, C \leftarrow H_3, D \leftarrow H_4, E \leftarrow H_5$
Round 1	For j=0 to 19 $t \leftarrow ((A \ll 5) + f(B, C, D) + E + Xj + y_1)$ $E \leftarrow D, D \leftarrow C, C \leftarrow (B \ll 30), B \leftarrow A, A \leftarrow t$
Round 2	For j=20 to 39 $t \leftarrow ((A \ll 5) + h(B, C, D) + E + Xj + y_2)$ $E \leftarrow D, D \leftarrow C, C \leftarrow (B \ll 30), B \leftarrow A, A \leftarrow t$
Round 3	For j=40 to 59 $t \leftarrow ((A \ll 5) + g(B, C, D) + E + Xj + y_3)$ $E \leftarrow D, D \leftarrow C, C \leftarrow (B \ll 30), B \leftarrow A, A \leftarrow t$
Round 4	For j=60 to 79 $t \leftarrow ((A \ll 5) + h(B, C, D) + E + Xj + y_4)$ $E \leftarrow D, D \leftarrow C, C \leftarrow (B \ll 30), B \leftarrow A, A \leftarrow t$
Update chaining variables	$H_1 \leftarrow H_1 + A, H_2 \leftarrow H_2 + B,$ $H_3 \leftarrow H_3 + C, H_4 \leftarrow H_4 + D,$ $H_5 \leftarrow H_5 + E$

The bold text is to emphasize the differences between each round.

10   ### 13.1.2.3 Step 3. Completion

After all the 512-bit blocks of the padded input message have been processed, the output hash value is the final 160-bit value given by: $H_1 | H_2 | H_3 | H_4 | H_5$.

13.1.2.4 Optimization for hardware implementation

15   The SHA-1 Step 2 procedure is not optimized for hardware. In particular, the 80 temporary 32-bit registers use up valuable silicon on a hardware implementation. This section describes an

695

optimization to the SHA-1 algorithm that only uses 16 temporary registers. The reduction in silicon is from 2560 bits down to 512 bits, a saving of over 2000 bits. It may not be important in some applications, but in the QA Chip storage space must be reduced where possible.

5 The optimization is based on the fact that although the original 16-word message block is expanded into an 80-word message block, the 80 words are not updated during the algorithm. In addition, the words rely on the previous 16 words only, and hence the expanded words can be calculated on-the-fly during processing, as long as we keep 16 words for the backward references. We require rotating counters to keep track of which register we are up to using, but the effect is to save a large
10 amount of storage.

Rather than index X by a single value j, we use a 5 bit counter to count through the iterations. This can be achieved by initializing a 5-bit register with either 16 or 20, and decrementing it until it reaches 0. In order to update the 16 temporary variables as if they were 80, we require 4 indexes,
15 each a 4-bit register. All 4 indexes increment (with wraparound) during the course of the algorithm.

Table 237. Optimised Steps to follow for each 512 bit block (InputWord$_{0-15}$)

Initialize working variables	$A \leftarrow H_1$, $B \leftarrow H_2$, $C \leftarrow H_3$, $D \leftarrow H_4$, $E \leftarrow H_5$
	$N_1 \leftarrow 13$, $N_2 \leftarrow 8$, $N_3 \leftarrow 2$, $N_4 \leftarrow 0$
Round 0 Copy the 512 input bits into X_{0-15}	Do 16 times $X_{N4} = \text{InputWord}_{N4}$ $[\Uparrow N_1, \Uparrow N_2, \Uparrow N_3]_{\text{optional}}\ \Uparrow N_4$
Round 1A	Do 16 times $t \leftarrow ((A \ll 5) + f(B, C, D) + E + X_{N4} + y_1)$ $[\Uparrow N_1, \Uparrow N_2, \Uparrow N_3]_{\text{optional}}\ \Uparrow N_4$ $E \leftarrow D$, $D \leftarrow C$, $C \leftarrow (B \ll 30)$, $B \leftarrow A$, $A \leftarrow t$
Round 1B	Do 4 times $X_{N4} \leftarrow ((X_{N1} \oplus X_{N2} \oplus X_{N3} \oplus X_{N4}) \ll 1)$ $t \leftarrow ((A \ll 5) + f(B, C, D) + E + X_{N4} + y_1)$ $\Uparrow N_1, \Uparrow N_2, \Uparrow N_3, \Uparrow N_4$ $E \leftarrow D$, $D \leftarrow C$, $C \leftarrow (B \ll 30)$, $B \leftarrow A$, $A \leftarrow t$
Round 2	Do 20 times $X_{N4} \leftarrow ((X_{N1} \oplus X_{N2} \oplus X_{N3} \oplus X_{N4}) \ll 1)$ $t \leftarrow ((A \ll 5) + h(B, C, D) + E + X_{N4} + y_2)$ $\Uparrow N_1, \Uparrow N_2, \Uparrow N_3, \Uparrow N_4$

	$E \leftarrow D, D \leftarrow C, C \leftarrow (B \ll 30), B \leftarrow A, A \leftarrow t$
Round 3	Do 20 times
	$X_{N4} \leftarrow ((X_{N1} \oplus X_{N2} \oplus X_{N3} \oplus X_{N4}) \ll 1)$
	$t \leftarrow ((A \ll 5) + g(B, C, D) + E + X_{N4} + y_3)$
	$\Uparrow N_1, \Uparrow N_2, \Uparrow N_3, \Uparrow N_4$
	$E \leftarrow D, D \leftarrow C, C \leftarrow (B \ll 30), B \leftarrow A, A \leftarrow t$
Round 4	Do 20 times
	$X_{N4} \leftarrow ((X_{N1} \oplus X_{N2} \oplus X_{N3} \oplus X_{N4}) \ll 1)$
	$t \leftarrow ((A \ll 5) + h(B, C, D) + E + X_{N4} + y_4)$
	$\Uparrow N_1, \Uparrow N_2, \Uparrow N_3, \Uparrow N_4$
	$E \leftarrow D, D \leftarrow C, C \leftarrow (B \ll 30), B \leftarrow A, A \leftarrow t$
Update chaining variables	$H_1 \leftarrow H_1 + A, H_2 \leftarrow H_2 + B,$
	$H_3 \leftarrow H_3 + C, H_4 \leftarrow H_4 + D,$
	$H_5 \leftarrow H_5 + E$

The bold text is to emphasize the differences between each round.

The incrementing of N_1, N_2, and N_3 during Rounds 0 and 1A is optional. A software implementation would not increment them, since it takes time, and at the end of the 16 times through the loop, all 4 counters will be their original values. Designers of hardware may wish to increment all 4 counters together to save on control logic.

Round 0 can be completely omitted if the caller loads the 512 bits of X_{0-15}.

14 Holding Out Against Attacks

The authentication protocols described in Section 7 on page 661 onward should be resistant to defeat by logical means. This section details each type of attack in turn with reference to the Read Authentication protocol.

14.1 BRUTE FORCE ATTACK

A brute force attack is guaranteed to break any protocol. However the length of the key means that the time for an attacker to perform a brute force attack is too long to be worth the effort.

An attacker only needs to break K to build a clone authentication chip. A brute force attack on K must therefore break a 160-bit key.

An attack against K requires a maximum of 2^{160} attempts, with a 50% chance of finding the key after only 2^{159} attempts. Assuming an array of a trillion processors, each running one million tests per second, 2^{159} (7.3×10^{47}) tests takes 2.3×10^{22} years, which is longer than the total lifetime of the universe. There are around 100 million personal computers in the world. Even if these were all connected in an attack (e.g. via the Internet), this number is still 10,000 times smaller than the trillion-processor attack described. Further, if the manufacture of one trillion processors becomes a possibility in the age of nanocomputers, the time taken to obtain the key is still longer than the total lifetime of the universe.

14.2   GUESSING THE KEY ATTACK

It is theoretically possible that an attacker can simply "guess the key". In fact, given enough time, and trying every possible number, an attacker will obtain the key. This is identical to the brute force attack described above, where 2^{159} attempts must be made before a 50% chance of success is obtained.

The chances of someone simply guessing the key on the first try is 2^{160}. For comparison, the chance of someone winning the top prize in a U.S. state lottery and being killed by lightning in the same day is only 1 in 2^{61} [78]. The chance of someone guessing the authentication chip key on the first go is 1 in 2^{160}, which is comparable to two people choosing exactly the same atoms from a choice of all the atoms in the Earth i.e. extremely unlikely.

14.3   QUANTUM COMPUTER ATTACK

To break K, a quantum computer containing 160 qubits embedded in an appropriate algorithm must be built. As described in Section 5.7.1.7 on page 648, an attack against a 160-bit key is not feasible. An outside estimate of the possibility of quantum computers is that 50 qubits may be achievable within 50 years. Even using a 50 qubit quantum computer, 2^{110} tests are required to crack a 160 bit key. Assuming an array of 1 billion 50 qubit quantum computers, each able to try 2^{50} keys in 1 microsecond (beyond the current wildest estimates) finding the key would take an average of 18 billion years.

14.4   CIPHERTEXT ONLY ATTACK

An attacker can launch a ciphertext only attack on K by monitoring calls to Random and Read. However, given that all these calls also reveal the plaintext as well as the hashed form of the plaintext, the attack would be transformed into a stronger form of attack - a known plaintext attack.

14.5   KNOWN PLAINTEXT ATTACK

It is easy to connect a logic analyzer to the connection between the System and the authentication chip, and thereby monitor the flow of data. This flow of data results in known plaintext and the hashed form of the plaintext, which can therefore be used to launch a known plaintext attack against K.

5   To launch an attack against K, multiple calls to Random and Test must be made (with the call to Test being successful, and therefore requiring a call to Read on a valid chip). This is straightforward, requiring the attacker to have both a system authentication chip and a consumable authentication chip. For each set of calls, an X, $S_K[X]$ pair is revealed. The attacker must collect these pairs for further analysis.

10 The question arises of how many pairs must be collected for a meaningful attack to be launched with this data. An example of an attack that requires collection of data for statistical analysis is differential cryptanalysis (see Section 14.13 on page 703). However, there are no known attacks against SHA-1 or HMAC-SHA1 [7][7][7], so there is no use for the collected data at this time.

15 14.6   CHOSEN PLAINTEXT ATTACKS

The golden rule for the QA Chip is that it never signs something that is simply given to it - i.e. it never lets the user choose the message that is signed.

Although the attacker can choose both R_T and possibly M, ChipA advances its random number R_A
20 with each call to Read. The resultant message X therefore contains 160 bits of changing data each call that are not chosen by the attacker.

To launch a chosen text attack the attacker would need to locate a chip whose R was the desired R. This makes the search effectively impossible.

25

14.7   ADAPTIVE CHOSEN PLAINTEXT ATTACKS

The HMAC construct provides security against all forms of chosen plaintext attacks [7]. This is primarily because the HMAC construct has 2 secret input variables (the result of the original hash, and the secret key). Thus finding collisions in the hash function itself when the input variable is
30 secret is even harder than finding collisions in the plain hash function. This is because the former requires direct access to SHA-1 in order to generate pairs of input/output from SHA-1.

Since R changes with each call to Read, the user cannot choose the complete message. The only value that can be collected by an attacker is $HMAC[R_1 \mid R_2 \mid M_2]$. These are not attacks against the
35 SHA-1 hash function itself, and reduce the attack to a differential cryptanalysis attack (see Section 14.13 on page 703), examining statistical differences between collected data. Given that there is no

differential cryptanalysis attack known against SHA-1 or HMAC, the protocols are resistant to the adaptive chosen plaintext attacks.

14.8   PURPOSEFUL ERROR ATTACK

5   An attacker can only launch a purposeful error attack on the Test function, since this is the only function in the Read protocol that validates input against the keys.

With the Test function, a 0 value is produced if an error is found in the input - no further information is given. In addition, the time taken to produce the 0 result is independent of the input, giving the
10   attacker no information about which bit(s) were wrong.
A purposeful error attack is therefore fruitless.

14.9   CHAINING ATTACK

Any form of chaining attack assumes that the message to be hashed is over several blocks, or the
15   input variables can somehow be set. The HMAC-SHA1 algorithm used by Protocol C1 only ever hashes one or two 512-bit blocks. Chaining attacks are not possible when only one block is used, and are extremely limited when two blocks are used.

14.10   BIRTHDAY ATTACK

20   The strongest attack known against HMAC is the birthday attack, based on the frequency of collisions for the hash function [7][7]. However this is totally impractical for minimally reasonable hash functions such as SHA-1. And the birthday attack is only possible when the attacker has control over the message that is hashed.
Since in the protocols described for the QA Chip, the message to be signed is never chosen by the
25   attacker (at least one 160-bit R value is chosen by the chip doing the signing), the attacker has no control over the message that is hashed. An attacker must instead search for a collision message that hashes to the same value (analogous to finding one person who shares your birthday).

The clone chip must therefore attempt to find a new value R_2 such that the hash of R_1, R_2 and a
30   chosen M_2 yields the same hash value as $H[R_1|R_2|M]$. However ChipT does not reveal the correct hash value (the Test function only returns 1 or 0 depending on whether the hash value is correct). Therefore the only way of finding out the correct hash value (in order to find a collision) is to interrogate a real ChipA. But to find the correct value means to update M, and since the decrement-only parts of M are one-way, and the read-only parts of M cannot be changed, a clone consumable
35   would have to update a real consumable before attempting to find a collision. The alternative is a brute force attack search on the Test function to find a success (requiring each clone consumable

700

to have access to a System consumable). A brute force search, as described above, takes longer than the lifetime of the universe, in this case, per authentication.

There is no point for a clone consumable to launch this kind of attack.

14.11 SUBSTITUTION WITH A COMPLETE LOOKUP TABLE

The random number seed in each System is 160 bits. The best case situation for an attacker is that no state data has been changed. Assuming also that the clone consumable does not advance its R, there is a constant value returned as M. A clone chip must therefore return $S_K[R \mid c]$ (where c is a constant), which is a 160 bit value.

Assuming a 160-bit lookup of a 160-bit result, this requires 2.9×10^{49} bytes, or 2.6×10^{37} terabytes, certainly more space than is feasible for the near future. This of course does not even take into account the method of collecting the values for the ROM. A complete lookup table is therefore completely impossible.

14.12 SUBSTITUTION WITH A SPARSE LOOKUP TABLE

A sparse lookup table is only feasible if the messages sent to the authentication chip are somehow predictable, rather than effectively random.

The random number R is seeded with an unknown random number, gathered from a naturally System authentication chip's Random function, and iterating some random event. There is no possibility for a clone manufacturer to know what the possible range of R is for all Systems, since each bit has an unrelated chance of being 1 or 0.

Since the range of R in all systems is unknown, it is not possible to build a sparse lookup table that can be used in all systems. The general sparse lookup table is therefore not a possible attack.

However, it is possible for a clone manufacturer to know what the range of R is for a given System. This can be accomplished by loading a LFSR with the current result from a call to a specific number of times into the future. If this is done, a special ROM can be built which will only contain the responses for that particular range of R, i.e. a ROM specifically for the consumables of that particular System. But the attacker still needs to place correct information in the ROM. The attacker will therefore need to find a valid authentication chip and call it for each of the values in R. Suppose the clone authentication chip reports a full consumable, and then allows a single use before simulating loss of connection and insertion of a new full consumable. The clone consumable would therefore need to contain responses for authentication of a full consumable and authentication of a partially used consumable. The worst case ROM contains entries for full and

partially used consumables for R over the lifetime of System. However, a valid authentication chip must be used to generate the information, and be partially used in the process. If a given System only produces n R-values, the sparse lookup-ROM required is $20n$ bytes ($20 = 160 / 8$) multiplied by the number of different values for M. The time taken to build the ROM depends on the amount of

5   time enforced between calls to Read.

After all this, the clone manufacturer must rely on the consumer returning for a refill, since the cost of building the ROM in the first place consumes a single consumable. The clone manufacturer's business in such a situation is consequently in the refills.

10   The time and cost then, depends on the size of R and the number of different values for M that must be incorporated in the lookup. In addition, a custom clone consumable ROM must be built to match each and every System, and a different valid authentication chip must be used for each System (in order to provide the full and partially used data). The use of an authentication chip in a System must therefore be examined to determine whether or not this kind of attack is worthwhile for

15   a clone manufacturer.

As an example, of a camera system that has about 10,000 prints in its lifetime. Assume it has a single Decrement Only value (number of prints remaining), and a delay of 1 second between calls to Read. In such a system, the sparse table will take about 3 hours to build, and consumes 100K.

20   Remember that the construction of the ROM requires the consumption of a valid authentication chip, so any money charged must be worth more than a single consumable and the clone consumable combined. Thus it is not cost effective to perform this function for a single consumable (unless the clone consumable somehow contained the equivalent of multiple authentic consumables).

25

If a clone manufacturer is going to go to the trouble of building a custom ROM for each owner of a System, an easier approach would be to update System to completely ignore the authentication chip.

30   Consequently, this attack is possible as a per-System attack, and a decision must be made about the chance of this occurring for a given System/Consumable combination. The chance will depend on the cost of the consumable and authentication chips, the longevity of the consumable, the profit margin on the consumable, the time taken to generate the ROM, the size of the resultant ROM, and whether customers will come back to the clone manufacturer for refills that use the same clone chip

35   etc.

14.13 DIFFERENTIAL CRYPTANALYSIS

Existing differential attacks are heavily dependent on the structure of S boxes, as used in DES and other similar algorithms. Although HMAC-SHA1 has no S boxes, an attacker can undertake a differential-like attack by undertaking statistical analysis of:

5
- Minimal-difference inputs, and their corresponding outputs
- Minimal-difference outputs, and their corresponding inputs

To launch an attack of this nature, sets of input/output pairs must be collected. The collection can be via known plaintext, or from a partially adaptive chosen plaintext attack. Obviously the latter,
10 being chosen, will be more useful.

Hashing algorithms in general are designed to be resistant to differential analysis. SHA-1 in particular has been specifically strengthened, especially by the 80 word expansion so that minimal differences in input will still produce outputs that vary in a larger number of bit positions (compared
15 to 128 bit hash functions). In addition, the information collected is not a direct SHA-1 input/output set, due to the nature of the HMAC algorithm. The HMAC algorithm hashes a known value with an unknown value (the key), and the result of this hash is then rehashed with a separate unknown value. Since the attacker does not know the secret value, nor the result of the first hash, the inputs and outputs from SHA-1 are not known, making any differential attack extremely difficult.
20

There are no known differential attacks against SHA-1 or HMAC-SHA-1[56][56].

The following is a more detailed discussion of minimally different inputs and outputs from the QA Chip.

25 ### 14.13.1 Minimal difference inputs

This is where an attacker takes a set of X, $S_K[X]$ values where the X values are minimally different, and examines the statistical differences between the outputs $S_K[X]$. The attack relies on X values that only differ by a minimal number of bits. The question then arises as to how to obtain minimally different X values in order to compare the $S_K[X]$ values.
30

Although the attacker can choose both R_T and possibly M, ChipA advances its random number R_A with each call to Read. The resultant X therefore contains 160 bits of changing data each call, and is therefore not minimally different.

14.13.2 Minimal difference outputs

This is where an attacker takes a set of X, $S_K[X]$ values where the $S_K[X]$ values are minimally different, and examines the statistical differences between the X values. The attack relies on $S_K[X]$ values that only differ by a minimal number of bits.

There is no way for an attacker to generate an X value for a given $S_K[X]$. To do so would violate the fact that S is a one-way function (HMAC-SHA1). Consequently the only way for an attacker to mount an attack of this nature is to record all observed X, $S_K[X]$ pairs in a table. A search must then be made through the observed values for enough minimally different $S_K[X]$ values to undertake a statistical analysis of the X values.

14.14 MESSAGE SUBSTITUTION ATTACKS

In order for this kind of attack to be carried out, a clone consumable must contain a real authentication chip, but one that is effectively reusable since it never gets decremented. The clone authentication chip would intercept messages, and substitute its own. However this attack does not give success to the attacker.

A clone authentication chip may choose not to pass on a Write command to the real authentication chip. However the subsequent Read command must return the correct response (as if the Write had succeeded). To return the correct response, the hash value must be known for the specific R and M. An attacker can only determine the hash value by actually updating M in a real Chip, which the attacker does not want to do. Even changing the R sent by System does not help since the System authentication chip must match the R during a subsequent Test.

A message substitution attack would therefore be unsuccessful. This is only true if System updates the amount of consumable remaining before it is used.

14.15 REVERSE ENGINEERING THE KEY GENERATOR

If a pseudo-random number generator is used to generate keys, there is the potential for a clone manufacture to obtain the generator program or to deduce the random seed used. This was the way in which the security layer of the Netscape browser was initially broken [33].

14.16 BYPASSING THE AUTHENTICATION PROCESS

The System should ideally update the consumable state data before the consumable is used, and follow every write by a read (to authenticate the write). Thus each use of the consumable requires an authentication. If the System adheres to these two simple rules, a clone manufacturer will have to simulate authentication via a method above (such as sparse ROM lookup).

14.17 REUSE OF AUTHENTICATION CHIPS

Each use of the consumable requires an authentication. If a consumable has been used up, then its authentication chip will have had the appropriate state-data values decremented to 0. The chip can therefore not be used in another consumable.

Note that this only holds true for authentication chips that hold Decrement-Only data items. If there is no state data decremented with each usage, there is nothing stopping the reuse of the chip. This is the basic difference between Presence-Only authentication and Consumable Lifetime authentication. All described protocols allow both.

The bottom line is that if a consumable has Decrement Only data items that are used by the System, the authentication chip cannot be reused without being completely reprogrammed by a valid programming station that has knowledge of the secret key (e.g. an authorized refill station).

14.18 MANAGEMENT DECISION TO OMIT AUTHENTICATION TO SAVE COSTS

Although not strictly an external attack, a decision to omit authentication in future Systems in order to save costs will have widely varying effects on different markets.

In the case of high volume consumables, it is essential to remember that it is very difficult to introduce authentication after the market has started, as systems requiring authenticated consumables will not work with older consumables still in circulation. Likewise, it is impractical to discontinue authentication at any stage, as older Systems will not work with the new, unauthenticated, consumables. In the second case, older Systems can be individually altered by replacing the System program code.

Without any form of protection, illegal cloning of high volume consumables is almost certain. However, with the patent and copyright protection, the probability of illegal cloning may be, say 50%. However, this is not the only loss possible. If a clone manufacturer were to introduce clone consumables which caused damage to the System (e.g. clogged nozzles in a printer due to poor quality ink), then the loss in market acceptance, and the expense of warranty repairs, may be significant.

In the case of a specialized pairing, such as a car/car-keys, or door/door-key, or some other similar situation, the omission of authentication in future systems is trivial and without repercussions. This is because the consumer is sold the entire set of System and Consumable authentication chips at the one time.

705

14.19 GARROTE/BRIBE ATTACK

If humans do not know the key, there is no amount of force or bribery that can reveal them. The use of ChipF and the ReplaceKey protocol is specifically designed to avoid the requirement of the programming station having to know the new key. However ChipF must be told the new key at some stage, and therefore it is the person(s) who enter the new key into ChipF that are at risk.

The level of security against this kind of attack is ultimately a decision for the System/Consumable owner, to be made according to the desired level of service.

For example, a car company may wish to keep a record of all keys manufactured, so that a person can request a new key to be made for their car. However this allows the potential compromise of the entire key database, allowing an attacker to make keys for any of the manufacturer's existing cars. It does not allow an attacker to make keys for any new cars. Of course, the key database itself may also be encrypted with a further key that requires a certain number of people to combine their key portions together for access. If no record is kept of which key is used in a particular car, there is no way to make additional keys should one become lost. Thus an owner will have to replace his car's authentication chip and all his car-keys. This is not necessarily a bad situation.

By contrast, in a consumable such as a printer ink cartridge, the one key combination is used for all Systems and all consumables. Certainly if no backup of the keys is kept, there is no human with knowledge of the key, and therefore no attack is possible. However, a no-backup situation is not desirable for a consumable such as ink cartridges, since if the key is lost no more consumables can be made. The manufacturer should therefore keep a backup of the key information in several parts, where a certain number of people must together combine their portions to reveal the full key information. This may be required if case the chip programming station needs to be reloaded.

In any case, none of these attacks are against the authenticated read protocol, since no humans are involved in the authentication process.

LOGICAL INTERFACE

15 Introduction

The QA Chip has a physical and a logical external interface. The physical interface defines how the QA Chip can be connected to a physical System, while the logical interface determines how that System can communicate with the QA Chip. This section deals with the logical interface.

15.1 OPERATING MODES

The QA Chip has four operating modes - *Idle Mode*, *Program Mode*, *Trim Mode* and *Active Mode*.

- *Idle Mode* is used to allow the chip to wait for the next instruction from the System.

- *Trim Mode* is used to determine the clock speed of the chip and to trim the frequency during the initial programming stage of the chip (when Flash memory is garbage). The clock frequency *must* be trimmed via Trim Mode *before* Program Mode is used to store the program code.

- *Program Mode* is used to load up the operating program code, and is required because the operating program code is stored in Flash memory instead of ROM (for security reasons).

- *Active Mode* is used to execute the specific authentication command specified by the System. Program code is executed in *Active Mode*. When the results of the command have been returned to the System, the chip enters *Idle Mode* to wait for the next instruction.

15.1.1   Idle Mode

The QA Chip starts up in *Idle Mode*. When the Chip is in *Idle Mode*, it waits for a command from the master by watching the primary id on the serial line.

- If the primary id matches the global id (0x00, common to all QA Chips), and the following byte from the master is the Trim Mode id byte, the QA Chip enters *Trim Mode* and starts counting the number of internal clock cycles until the next byte is received.

- If the primary id matches the global id (0x00, common to all QA Chips), and the following byte from the master is the Program Mode id byte, the QA Chip enters *Program Mode*.

- If the primary id matches the global id (0x00, common to all QA Chips), and the following byte from the master is the Active Mode id byte, the QA Chip enters *Active Mode* and executes startup code, allowing the chip to set itself into a state to receive authentication commands (includes setting a local id).

- If the primary id matches the chip's local id, and the following byte is a valid command code, the QA Chip enters *Active Mode*, allowing the command to be executed.

The valid 8-bit serial mode values sent after a global id are as shown in Table 238. They are specified to minimize the chances of them occurring by error after a global id (e.g. 0xFF and 0x00 are not used):

Table 238. Id byte values to place chip in specific mode

Value	Interpretation
10100101 (0xA5)	Trim Mode
10001110 (0x8E)	Program Mode
01111000 (0x78)	Active Mode

15.1.2   Trim Mode

Trim Mode is enabled by sending a global id byte (0x00) followed by the Trim Mode command byte. The purpose of Trim Mode is to set the trim value (an internal register setting) of the internal ring oscillator so that Flash erasures and writes are of the correct duration. This is necessary due to the variation of the clock speed due to process variations. If writes an erasures are too long, the Flash memory will wear out faster than desired, and in some cases can even be damaged.

Trim Mode works by measuring the number of system clock cycles that occur inside the chip from the receipt of the Trim Mode command byte until the receipt of a data byte. When the data byte is received, the data byte is copied to the trim register and the current value of the count is transmitted to the outside world.

Once the count has been transmitted, the QA Chip returns to *Idle Mode*.

At reset, the internal trim register setting is set to a known value r. The external user can now perform the following operations:

- send the global id+write followed by the Trim Mode command byte
- send the 8-bit value v over a specified time t
- send a stop bit to signify no more data
- send the global id+read followed by the Trim Mode command byte
- receive the count c
- send a stop bit to signify no more data

At the end of this procedure, the trim register will be v, and the external user will know the relationship between external time t and internal time c. Therefore a new value for v can be calculated.

The Trim Mode procedure can be repeated a number of times, varying both t and v in known ways, measuring the resultant c. At the end of the process, the final value for v is established (and stored in the trim register for subsequent use in Program Mode). This value v must also be written to the flash for later use (every time the chip is placed in Active Mode for the first time after power-up).

15.1.3   Program Mode

Program Mode is enabled by sending a global id byte (0x00) followed by the Program Mode command byte.

The QA Chip determines whether or not the internal fuse has been blown (by reading 32-bit word 0 of the information block of flash memory).

If the fuse has been blown the Program Mode command is ignored, and the QA Chip returns to *Idle Mode*.

5   If the fuse is still intact, the chip enters Program Mode and erases the entire contents of Flash memory. The QA Chip then validates the erasure. If the erasure was successful, the QA Chip receives up to 4096 bytes of data corresponding to the new program code and variable data. The bytes are transferred in order $byte_0$ to $byte_{4095}$.

Once all bytes of data have been loaded into Flash, the QA Chip returns to *Idle Mode*.

10

Note that Trim Mode functionality *must be performed before* a chip enters Program Mode for the first time.

Once the desired number of bytes have been downloaded in Program Mode, the LSS Master must

15   wait for 80μs (the time taken to write two bytes to flash at nybble rates) before sending the new transaction (eg Active Mode). Otherwise the last nybbles may not be written to flash.

15.1.4   Active Mode

Active Mode is entered either by receiving a global id byte (0x00) followed by the Active Mode

20   command byte, or by sending a local id byte followed by a command opcode byte and an appropriate number of data bytes representing the required input parameters for that opcode.

In both cases, Active Mode causes execution of program code previously stored in the flash memory via Program Mode. As a result, we never enter Active Mode after Trim Mode, without a

25   Program Mode in between. However once programmed via Program Mode, a chip is allowed to enter Active Mode after power-up, since valid data will be in flash.

If Active Mode is entered by the global id mechanism, the QA Chip executes specific reset startup code, typically setting up the local id and other IO specific data.

30

If Active Mode is entered by the local id mechanism, the QA Chip executes specific code depending on the following byte, which functions as an opcode. The opcode command byte format is shown in Table 239:

Table 239. Command byte

35

bits	Description
2-0	Opcode

709

5-3	¬opcode
7-6	count of number of bits set in opcode (0 to 3)

The interpretation of the 3-bit opcode is shown in Table 240:

Table 240. QA Chip opcodes

Op[2]	Mn[3]	Description
000	RST	Reset
001	RND	Random
010	RDM	Read M
011	TST	Test
100	WRM	Write M with no authentication
101	WRA	Write with Authentication (to M, P, or K)
110		chip specific - reserved for ChipF, ChipS etc
111		chip specific - reserved for ChipF, ChipS etc

5

The command byte is designed to ensure that errors in transmission are detected. Regular QA Chip commands are therefore comprised of an opcode plus any associated parameters. The commands are listed in Table 241:

Table 241. QA Chip commands

10

Command	Input		Output
	opcode	Additional parms	Return value
Reset	RST	-	-
Random	RND	-	[20]
Read	RDM	[1, 1, 20]	[20, 64, 20][4]
Test	TST	[1, 20, 64, 20]	89[5] if successful, 76 if not
Write	WRM	[1, 64, 20]	89 if successful, 76 if not

---

[2]   Opcode

[3]   Mnemonic

[4]   [n, m] = list of parameters where n bytes for first parameter, and m bytes for the second etc.

[5]   n = actual byte pattern required (in hex). The bytes 0x76 and 0x89 were chosen as the bool ean values 0 and 1 as they are inverses of each other, and should not be generated acciden tally.

WriteAuth	WRA	76 [20, 64, 20]	89 if successful, 76 if not
ReplaceKey	WRA	89 76 [1, 20, 20, 20]	89 if successful, 76 if not
SetPermissions	WRA	89 89 [1, 1, 20, 4, 20]	[4]
SignM[6]	ChipS only	[1, 20, 20, 64, 20, 64]	[20, 64, 20]
SignP[7]	ChipS only	[1, 20, 20, 4, 20, 4]	[20, 64, 20]
GetProgKey	ChipF only	[1, 20]	[20, 20, 20]
SetPartialKey	ChipF only	[1, 4]	89 if successful, 76 if not

Apart from the Reset command, the next four commands are the commands most likely to be used during regular operation. The next three commands are used to provide authenticated writes (which are expected to be uncommon). The final set of commands (including SignM), are expected to be specially implemented on ChipS and ChipF QA Chips only.

The input parameters are sent in the specified order, with each parameter being sent least significant byte first and most significant byte last.

Return (output) values are read in the same way - least significant byte first and most significant byte last. The client must know how many bytes to retrieve. The QA Chip will time out and return to *Idle Mode* if an incorrect number of bytes is provided or read.

In most cases, the output bytes from one chip's command (the return values) can be fed directly as the input bytes to another chip's command. An example of this is the RND and RD commands. The output data from a call to RND on a trusted QA Chip does not have to be kept by the System. Instead, the System can transfer the output bytes directly to the input of the non-trusted QA Chip's RD command. The description of each command points out where this is so.

Each of the commands is examined in detail in the subsequent sections. Note that some algorithms are specifically designed because flash memory is assumed for the implementation of non-volatile variables.

15.1.5   Non volatile variables

The memory within the QA Chip contains some *non-volatile* (Flash) memory to store the variables required by the authentication protocol. Table 242 summarizes the variables.

Table 242. Non volatile variables required by the authentication protocol

Name	Size	Description

---

[6]   It is expected that most QA Chips will implement SignM as a function that returns 0x00. Only a limited number of chips will be programmed to allow SignM functionality. It is included here as an example of how signatures can be generated for authenticated writes.

[7]   It is expected that most QA Chips will implement SignP as a function that returns 0x00. Only a limited number of chips will be programmed to allow SignP functionality. It is included here as an example of how signatures can be generated for authenticated writes.

	(bits)	
N	8	Number of keys known to the chip
T	8	Number of vectors M is broken into
K_n R_K	160 per key, 160 for R_K	Array of N secret keys used for calculating $F_{Kn}[X]$ where K_n is the nth element of the array. Each K_n must not be stored directly in the QA Chip. Instead, each chip needs to store a single random number R_K (different for each chip), $K_n \oplus R_K$, and $\neg K_n \oplus R_K$. The stored $K_n \oplus R_K$ can be XORed with R_K to obtain the real K_n. Although $\neg K_n \oplus R_K$ must be stored to protect against differential attacks, it is not used.
R	160	Current random number used to ensure time varying messages. Each chip instance must be seeded with a different initial value. Changes for each signature generation.
M_T	512 per M	Array of T memory vectors. Only M_0 can be written to with an authorized write, while all Ms can be written to in an unauthorized write. Writes to M_0 are optimized for Flash usage, while updates to any other M_n are expensive with regards to Flash utilization, and are expected to be only performed once per section of M_n. M_1 contains T and N in ReadOnly form so users of the chip can know these two values.
P_{T+N}	32 per P	T+N element array of access permissions for each part of M. Entries n={0... T-1} hold access permissions for non-authenticated writes to M_n (no key required). Entries n={T to T+N-1}hold access permissions for authenticated writes to M_0 for K_n. Permission choices for each part of M are Read Only, Read/Write, and Decrement Only
MinTicks	32	The minimum number of clock ticks between calls to key-based functions.

Note that since these variables are in Flash memory, writes should be minimized. The it is not a simple matter to write a new value to replace the old. Care must be taken with flash endurance, and speed of access. This has an effect on the algorithms used to change Flash memory based registers. For example, Flash memory should not be used as a shift register.

A reset of the QA Chip has no effect on the non-volatile variables.

15.1.5.1 M and P

M_n contains application specific state data, such as serial numbers, batch numbers, and amount of consumable remaining. M_n can be read using the Read command and written to via the Write and WriteA commands.

M_0 is expected to be updated frequently, while each part of M_{1-n} should only be written to once. Only M_0 can be written to via the WriteA command.

M_1 contains the operating parameters of the chip as shown in Table 243, and M_{2-n} are application specific.

Table 243. Interpretation of M_1

Length	Bits	interpretation
8	7-0	Number of available keys
8	15-8	Number of available M vectors
16	31-16	Revision of chip
96	127-32	Manufacture id information
128	255-128	Serial number
8	263-256	Local id of chip
248	511-264	reserved

Each M_n is 512 bits in length, and is interpreted as a set of 16 × 32-bit words. Although M_n may contain a number of different elements, each 32-bit word differs only in write permissions. Each 32-bit word can always be read. Once in client memory, the 512 bits can be interpreted in any way chosen by the client. The different write permissions for each P are outlined in Table 244:

Table 244. Write permissions

Data type	permission description
Read Only	Can *never* be written to
ReadWrite	Can *always* be written to
Decrement Only	Can only be written to if the new value is less than the old value. Decrement Only values can be any multiple of 32 bits.

To accomplish the protection required for writing, a 2-bit permission value P is defined for each of the 32-bit words. Table 245 defines the interpretation of the 2-bit permission bit-pattern:

713

Table 245. Permission bit interpretation

Bits	Op	Interpretation	Action taken during Write command
00	RW	ReadWrite	The new 32-bit value is always written to M[n].
01	MSR	Decrement Only (Most Significant Region)	The new 32-bit value is only written to M[n] if it is less than the value currently in M[n]. This is used for access to the Most Significant 16 bits of a Decrement Only number.
10	NMSR	Decrement Only (Not the Most Significant Region)	The new 32-bit value is only written to M[n] if M[n-1] could also be written. The NMSR access mode allows multiple precision values of 32 bits and more (multiples of 32 bits) to decrement.
11	RO	Read Only	The new 32-bit value is ignored. M[n] is left unchanged.

The 16 sets of permission bits for each 512 bits of M are gathered together in a single 32-bit variable P, where bits $2n$ and $2n+1$ of P correspond to word n of M as follows:

Each 2-bit value is stored as a pair with the msb in bit 1, and the lsb in bit 0. Consequently, if words 0 to 5 of M had permission MSR, with words 6-15 of M permission RO, the 32-bit P variable would be 0xFFFFF555:

11-11-11-11-11-11-11-11-11-11-01-01-01-01-01-01

During execution of a Write and WriteA command, the appropriate Permissions[n] is examined for each M[n] starting from n=15 (msw of M) to n=0 (lsw of M), and a decision made as to whether the new M[n] value will replace the old. Note that it is important to process the M[n] from msw to lsw to correctly interpret the access permissions.

Permissions are set and read using the QA Chip's SetPermissions command. The default for P is all 0s (RW) with the exception of certain parts of M_1.

Note that the Decrement Only comparison is *unsigned*, so any Decrement Only values that require negative ranges must be shifted into a positive range. For example, a consumable with a

714

Decrement Only data item range of -50 to 50 must have the range shifted to be 0 to 100. The System must then interpret the range 0 to 100 as being -50 to 50. Note that most instances of Decrement Only ranges are N to 0, so there is no range shift required.

5   For Decrement Only data items, arrange the data in order *from most significant to least significant* 32-bit quantities from M[n] onward. The access mode for the most significant 32 bits (stored in M[n]) should be set to MSR. The remaining 32-bit entries for the data should have their permissions set to NMSR.

10   If erroneously set to NMSR, with no associated MSR region, each NMSR region will be considered independently instead of being a multi-precision comparison.

Examples of allocating M and Permission bits can be found in [86].

15   *15.1.5.2   K and R_K*

K is the 160-bit secret key used to protect M and to ensure that the contents of M are valid (when M is read from a non trusted chip). K is initially programmed after manufacture, and from that point on, K can only be updated to a new value if the old K is known. Since K must be kept secret, there is no command to directly read it.

20

K is used in the keyed one-way hash function HMAC-SHA1. As such it should be programmed with a *physically generated* random number, gathered from a physically random phenomenon. *K must NOT be generated with a computer-run random number generator.* The security of the QA Chips depends on K being generated in a way that is not deterministic.

25

Each K_n must not be stored directly in the QA Chip. Instead, each chip needs to store a single random number R_K (different for each chip), $K_n \oplus R_K$, and $\neg K_n \oplus R_K$. The stored $K_n \oplus R_K$ can be XORed with R_K to obtain the real K_n. Although $\neg K_n \oplus R_K$ must be stored to protect against differential attacks, it is not used.

30

15.1.5.3   R

R is a 160-bit random number seed that is set up after manufacture (when the chip is programmed) and from that point on, cannot be changed. R is used to ensure that each signed item contains time varying information (not chosen by an attacker), and each chip's R is unrelated from one chip to the 35   next.

R is used during the Test command to ensure that the R from the previous call to Random was used as the session key in generating the signature during Read. Likewise, R is used during the WriteAuth command to ensure that the R from the previous call to Read was used as the session key during generation of the signature in the remote Authenticated chip.

5

The only invalid value for R is 0. This is because R is changed via a 160-bit maximal period LFSR (Linear Feedback Shift Register) with taps on bits 0, 2, 3, and 5, and is changed only by a successful call to a signature generating function (e.g. Test, WriteAuth).

10 The logical security of the QA Chip relies not only upon the randomness of K and the strength of the HMAC-SHA1 algorithm. To prevent an attacker from building a sparse lookup table, the security of the QA Chip also depends on the range of R over the lifetime of *all* Systems. What this means is that an attacker must not be able to deduce what values of R there are in produced and future Systems. Ideally, R should be programmed with a *physically generated* random number, gathered

15 from a physically random phenomenon (must not be deterministic). *R must NOT be generated with a computer-run random number generator.*

15.1.5.4 MinTicks

There are two mechanisms for preventing an attacker from generating multiple calls to key-based

20 functions in a short period of time. The first is an internal ring oscillator that is temperature-filtered. The second mechanism is the 32-bit MinTicks variable, which is used to specify the minimum number of QA Chip clock ticks that must elapse between calls to key-based functions.

The MinTicks variable is set to a fixed value when the QA Chip is programmed. It could possibly be

25 stored in M_1.

The effective value of MinTicks depends on the *operating* clock speed and the notion of what constitutes a reasonable time between key-based function calls (application specific). The duration of a single tick depends on the operating clock speed. This is the fastest speed of the ring oscillator

30 generated clock (i.e. at the lowest valid operating temperature).

Once the duration of a tick is known, the MinTicks value can to be set. The value for MinTicks will be the minimum number of ticks required to pass between calls to the key-based functions (there is no need to protect Random as this produces the same output each time it is called multiple times in a

35 row). The value is a real-time number, and divided by the length of an operating tick.

It should be noted that the MinTicks variable *only slows down* an attacker and causes the attack to cost more since it does not stop an attacker using multiple System chips in parallel.

15.1.6 GetProgramKey

Input: n, R_E = [1 byte, 20 bytes]

Output: R_L, $E_{Kx}[S_{Kn}[R_E|R_L|C_3]]$, $S_{Kx}[R_L|E_{Kx}[S_{Kn}[R_E|R_L|C_3]|C_3]$ = [20, 20, 20]

Changes: R_L

Note: The GetProgramKey command is only implemented in ChipF, and not in all QA Chips.

The *GetProgramKey* command is used to produce the bytestream required for updating a specified key in ChipP. Only an QA Chip programmed with the correct values of the old K_n can respond correctly to the GetProgramKey request. The output bytestream from the Random command can be fed as the input bytestream to the ReplaceKey command on the QA Chip being programmed (ChipP).

The input bytestream consists of the appropriate opcode followed by the desired key to generate the signature, followed by 20 bytes of R_E(representing the random number read in from ChipP).

The local random number R_L is advanced, and signed in combination with R_E and C_3 by the chosen key to generate a time varying secret number known to both ChipF and ChipP. This signature is then XORed with the new key K_x (this encrypts the new key). The first two output parameters are signed with the old key to ensure that ChipP knows it decoded K_x correctly.

This whole procedure should only be allowed a given number of times. The actual number can conveniently be stored in the local $M_0[0]$ (eg word 0 of M_0) with ReadOnly permission. Of course another chip could perform an Authorised write to update the number (via a ChipS) should it be desired.

The GetProgramKey command is implemented by the following steps:

```
Loop through all of Flash, reading each word (will trigger checks)
Accept n
Restrict n to N
Accept R_E
If (M_0[0] = 0)
    Output 60 bytes of 0x00 # no more keys allowed to be generated
from this chipF
    Done
EndIf
```

```
          Advance R_L
          SIG ← S_Kn[R_L|R_E|C_3]  # calculation must take constant time
          Tmp ← SIG ⊕ K_X
5         Output R_L
          Output Tmp
          Decrement M_0[0]    #  reduce   the   number   of   allowable   key
          generations by 1
          SIG ← S_KX[R_L|Tmp|C_3]   # calculation must take constant time
10        Output SIG
```

15.1.7   Random

> Input: None
>
> Output: R_L = [20 bytes]
>
> Changes: None

15   The *Random* command is used by a client to obtain an input for use in a subsequent authentication
procedure. Since the Random command requires no input parameters, it is therefore simply 1 byte
containing the RND opcode.

The output of the Random command from a trusted QA Chip can be fed straight into the non-
20   trusted chip's Read command as part of the input parameters. There is no need for the client to
store them at all, since they are not required again. However the Test command will only succeed if
the data passed to the Read command was obtained first from the Random command.

If a caller only calls the Random function multiple times, the same output will be returned each time.
25   R will only advance to the next random number in the sequence after a successful call to a function
that returns or tests a signature (e.g. Test, see Section 15.1.13 on page 725 for more information).

The Random command is implemented by the following steps:
```
          Loop through all of Flash, reading each word (will trigger checks)
30        Output R_L
```

15.1.8   Read

> Input: n, t, R_E = [1 byte, 1 byte, 20 bytes]
>
> Output: R_L, M_{Lt}, $S_{Kn}[R_E|R_L|C_1|M_{Lt}]$ = [20 bytes, 64 bytes, 20 bytes]
>
35 Changes: R_L

The *Read* command is used to read the entire state data (M_t) from an QA Chip. Only an QA Chip
programmed with the correct value of K_n can respond correctly to the Read request. The output

bytestream from the Read command can be fed as the input bytestream to the Test command on a trusted QA Chip for verification, with M_t stored for later use if Test returns success.

5 The input bytestream consists of the RD opcode followed by the key number to use for the signature, which M to read, and the bytes 0-19 of R_E. 23 bytes are transferred in total. R_E is obtained by calling the trusted QA Chip's Random command. The 20 bytes output by the trusted chip's Random command can therefore be fed directly into the non-trusted chip's Read command, with no need for these bits to be stored by System.

10 Calls to Read must wait for MinTicksRemaining to reach 0 to ensure that a minimum time will elapse between calls to Read.

The output values are calculated, MinTicksRemaining is updated, and the signature is returned. The contents of M_{Lt} are transferred least significant byte to most significant byte. The signature
15 $S_{Kn}[R_E|R_L|C_1|M_{Lt}]$ must be calculated in constant time.

The next random number is generated from R using a 160-bit maximal period LFSR (tap selections on bits 5, 3, 2, and 0). The initial 160-bit value for R is set up when the chip is programmed, and can be any random number except 0 (an LFSR filled with 0s will produce a never-ending stream of 0s).
20 R is transformed by XORing bits 0, 2, 3, and 5 together, and shifting all 160 bits right 1 bit using the XOR result as the input bit to b_{159}. The process is shown in Figure 347 below.

Care should be taken when updating R since it lives in Flash. Program code must assume power could be removed at any time.
25

The Read command is implemented with the following steps:

```
            Wait for MinTicksRemaining to become 0
            Loop through all of Flash, reading each word (will trigger checks)
            Accept n
30          Accept t
            Restrict n to N
            Restrict t to T
            Accept R_E
            Advance R_L
35          Output R_L
            Output M_Lt
            Sig ← S_Kn[R_E|R_L|C_1|M_Lt] # calculation must take constant time
```

719

```
MinTicksRemaining ← MinTicks
Output Sig
Wait for MinTicksRemaining to become 0
```

5   15.1.9   Set Permissions

    Input: n, p, R_E, P_E, SIG_E = [1 byte, 1 byte, 20 bytes, 4 bytes, 20 bytes]
    Output: P_p
    Changes: P_p, R_L

10   The *SetPermissions* command is used to securely update the contents of P_p (containing QA Chip permissions). The WriteAuth command only attempts to replace P_p if the new value is signed combined with our local R.

It is only possible to sign messages by knowing K_n. This can be achieved by a call to the SignP
15   command (because only a ChipS can know K_n). It means that without a chip that can be used to produce the required signature, a write of any value to P_p is not possible.

The process is very similar to Test, except that if the validation succeeds, the P_E input parameter is additionally ORed with the current value for P_p. Note that this is an OR, and not a replace. Since the
20   SetParms command only sets bits in P_p, the effect is to allow the permission bits corresponding to M[n] to progress from RW to either MSR, NMSR, or RO.

The SetPermissions command is implemented with the following steps:
```
        Wait for MinTicksRemaining to become 0
25      Loop through all of Flash, reading each word (will trigger checks)

        Accept n
        Restrict n to N
        Accept p
30      Restrict p to T+N
        Accept R_E
        Accept P_E
        SIG_L ← S_Kn[R_L|R_E|P_E|C_2] # calculation must take constant time
        Accept SIG_E
35      If (SIG_E = SIG_L)
            Update R_L
            P_P ← P_P ∨ P_E
```

720

```
            EndIf
            Output P_P # success or failure will be determined by receiver
            MinTicksRemaining ← MinTicks
```

15.1.10 ReplaceKey

Input: n, R_E, V, SIG_E = [1 byte, 20 bytes, 20 bytes, 20 bytes]

Output: Boolean ($0x76$=failure, $0x89$ = success)

Changes:K_n, M_L, R_L

The *ReplaceKey* command is used to replace the specified key in the QA Chip flash memory. However K_n can only be replaced if the previous value is known. A return byte of 0x89 is produced if the key was successfully updated, while 0x76 is returned for failure.

A ReplaceKey command consists of the WRA command opcode followed by 0x89, 0x76, and then the appropriate parameters. Note that the new key is not sent in the clear, it is sent encrypted with the signature of R_L, R_E and C_3 (signed with the old key). The first two input parameters must be verified by generating a signature using the old key.

The ReplaceKey command is implemented with the following steps:
```
            Loop through all of Flash, reading each word (will trigger checks)
            Accept n
            Restrict n to N
            Accept R_E # session key from ChipF
            Accept V # encrypted key

            SIG_L ← S_Kn[R_E|V|C_3]  # calculation must take constant time
            Accept SIG_E
            If (SIG_L = SIG_E2) # comparison must take constant time
                SIG_L ← S_Kn[R_L|R_E|C_3] # calculation must take constant time
                Advance R_L
                K_E ← SIG_L ⊕ V
                K_n ← K_E                # involves storing (K_E ⊕ R_K) and (¬K_E ⊕
            R_K)
                Output 0x89 # success
            Else
                Output 0x76 # failure
            EndIf
```

15.1.11 SignM

Input: n,R_X,R_E,M_E,SIG_E,$M_{desired}$ = [1 byte, 20 bytes, 20 bytes, 64 bytes,32 bytes]

Output: R_L, M_{new}, $S_{Kn}[R_E \mid R_L \mid C_1 \mid M_{new}]$ = [20 bytes, 64 bytes, 20 bytes]

Changes: R_L

Note: The SignM command is only implemented in ChipS, and not in all QA Chips.

The *SignM* command is used to produce a valid signed M for use in an authenticated write transaction. Only an QA Chip programmed with correct value of K_n can respond correctly to the SignM request. The output bytestream from the SignM command can be fed as the input bytestream to the WriteA command on a different QA Chip.

The input bytestream consists of the SMR opcode followed by 1 byte containing the key number to use for generating the signature, 20 bytes of R_X (representing the number passed in as R to ChipU's READ command, i.e. typically 0), the output from the READ command (namely R_E, M_E, and SIG_E), and finally the desired M to write to ChipU.

The SignM command only succeeds when $SIG_E = S_K[R_X \mid R_E \mid C_1 \mid M_E]$, indicating that the request was generated from a chip that knows K. This generation and comparison *must take the same amount of time regardless of whether the input parameters are correct or not*. If the times are not the same, an attacker can gain information about which bits of the supplied signature are incorrect. If the signatures match, then R_L is updated to be the next random number in the sequence.

Since the SignM function generates signatures, the function must wait for the MinTicksRemaining register to reach 0 before processing takes place.

Once all the inputs have been verified, a new memory vector is produced by applying a specially stored P value (eg word 1 of M_0) and $M_{desired}$ against M_E. Effectively, it is performing a regular Write, but with separate P against someone else's M. The M_{new} is signed with an updated R_L (and the passed in R_E), and all three values are output (the random number R_L, M_{new}, and the signature). The time taken to generate this signature must be the same regardless of the inputs.

Typically, the SignM command will be acting as a form of consumable command, so that a given ChipS can only generate a given number of signatures. The actual number can conveniently be stored in M_0 (eg word 0 of M_0) with ReadOnly permissions. Of course another chip could perform an Authorised write to update the number (using another ChipS) should it be desired.

The SignM command is implemented with the following steps:

```
Wait for MinTicksRemaining to become 0
Loop through all of Flash, reading each word (will trigger checks)

Accept n
Restrict n to N
```

```
        Accept R_X                          # don't care what this number is
        Accept R_E
        Accept M_E
        SIG_L ← S_Kn[R_X|R_E|C_1|M_E] # calculation must take constant time
 5      Accept SIG_E
        Accept M_desired
        If ((SIG_E ≠ SIG_L) OR (M_L[0] = 0))  # fail if bad signature or if
        allowed sigs = 0
           Output appropriate number of 0       # report failure
10         Done
        EndIf


        Update R_L


15      # Create the new version of M in ram from W and Permissions
        # This is the same as the core process of Write function
        # except that we don't write the results back to M
        DecEncountered ← 0
        EqEncountered ← 0
20      Permissions = M_L[1]                          #      assuming      M_0
        contains appropriate permissions
        For n ← msw to lsw #(word 15 to 0)
           AM ← Permissions[n]
           LT ← (M_desired[n] < M_E[n]) # comparison is unsigned
25         EQ ← (M_desired[n] = M_E[n])
           WE ← (AM = RW) ∨ ((AM = MSR) ∧ LT) ∨ ((AM = NMSR) ∧
        (DecEncountered ∨ LT))
           DecEncountered ← ((AM = MSR) ∧ LT)
                        ∨ ((AM = NMSR) ∧ DecEncountered)
30                      ∨ ((AM = NMSR) ∧ EqEncountered ∧ LT)
           EqEncountered ← ((AM = MSR) ∧ EQ) ∨ ((AM = NMSR) ∧
        EqEncountered ∧ EQ)
            If (¬WE) ∧ (M_E[n] ≠ M_desired[n])
              Output appropriate number of 0       # report failure
35          EndIf
        EndFor
```

723

```
                 # At this point, M_desired is correct
                 Output R_L
                 Output M_desired              # M_desired is now effectively M_new
5                Sig ← S_Kn[R_E|R_L|C_1|M_desired]   # calculation must take constant time
                 MinTicksRemaining ← MinTicks
                 Decrement M_L[0]     # reduce the number of allowable signatures by
                 1
                 Output Sig
```

10   15.1.12 SignP

Input: $n, R_E, P_{desired}$ = [1 byte, 20 bytes, 4 bytes]

Output: $R_L, S_{Kn}[R_E \mid R_L \mid P_{desired}|C_2]$ = [20 bytes, 20 bytes]

Changes: R_L

Note: The SignP command is only implemented in ChipS, and not in all QA Chips.

15

The *SignP* command is used to produce a valid signed P for use in a SetPermissions transaction. Only an QA Chip programmed with correct value of K_n can respond correctly to the SignP request. The output bytestream from the SignP command can be fed as the input bytestream to the SetPermissions command on a different QA Chip.

20

The input bytestream consists of the SMP opcode followed by 1 byte containing the key number to use for generating the signature, 20 bytes of R_E (representing the number obtained from ChipU's RND command, and finally the desired P to write to ChipU.

25   Since the SignP function generates signatures, the function must wait for the MinTicksRemaining register to reach 0 before processing takes place.

Once all the inputs have been verified, the $P_{desired}$ is signed with an updated R_L (and the passed in R_E), and both values are output (the random number R_L and the signature). The time taken to
30   generate this signature must be the same regardless of the inputs.

Typically, the SignP command will be acting as a form of consumable command, so that a given ChipS can only generate a given number of signatures. The actual number can conveniently be stored in M_0 (eg word 0 of M_0) with ReadOnly permissions. Of course another chip could perform an
35   Authorised write to update the number (using another ChipS) should it be desired.

The SignM command is implemented with the following steps:

```
        Wait for MinTicksRemaining to become 0
        Loop through all of Flash, reading each word (will trigger checks)

        Accept n
        Restrict n to N
        Accept R_E
        Accept P_desired
        If (M_L[0] = 0) # fail if allowed sigs = 0
           Output appropriate number of 0      # report failure
              Done
        EndIf

        Update R_L
        Output R_L
        Sig ← S_Kn[R_E|R_L|P_desired|C_2]   # calculation must take constant time
        MinTicksRemaining ← MinTicks
        Decrement M_L[0]    # reduce the number of allowable signatures by
        1
        Output Sig
```

15.1.13 Test

> Input: n, R_E, M_E, SIG$_E$ = [1 byte, 20 bytes, 64 bytes, 20 bytes]
> Output: Boolean (0x76=failure, 0x89 = success)
> Changes: R_L

The *Test* command is used to authenticate a read of an M from a non-trusted QA Chip.

The Test command consists of the TST command opcode followed by input parameters: n, R_E, M_E, and SIG$_E$. The byte order is least significant byte to most significant byte for each command component. All but the first input parameter bytes are obtained as the output bytes from a Read command to a non-trusted QA Chip. The entire data does not have to be stored by the client. Instead, the bytes can be passed directly to the trusted QA Chip's Test command, and only M should be kept from the Read.

Calls to Test must wait for the MinTicksRemaining register to reach 0.
$S_{Kn}[R_L|R_E|C_1|M_E]$ is then calculated, and compared against the input signature SIG$_E$. If they are different, R_L is not changed, and 0x76 is returned to indicate failure. If they are the same, then R_L is

updated to be the next random number in the sequence and 0x89 is returned to indicate success. Updating R_L only after success forces the caller to use a new random number (via the Random command) each time a successful authentication is performed.

5   The calculation of $S_{Kn}[R_L|R_E|C_1|M_E]$ and the comparison against SIG_E must take identical time so that *the time to evaluate the comparison in the TST function is always the same.* Thus no attacker can compare execution times or number of bits processed before an output is given.

    The Test command is implemented with the following steps:

10
```
        Wait for MinTicksRemaining to become 0
        Loop through all of Flash, reading each word (will trigger checks)

        Accept n
        Restrict n to N
15      Accept R_E
        Accept M_E
        SIG_L ← S_Kn[R_L|R_E|C_1|M_E] # calculation must take constant time
        Accept SIG_E
        If (SIG_E = SIG_L)
20         Update R_L
           Output 0x89 # success
        Else
           Output 0x76                 # report failure
        EndIf
25      MinTicksRemaining ← MinTicks
```

15.1.14   Write

         Input: t, M_{new}, SIG_E = [1 byte, 64 bytes, 20 bytes]

         Output: Boolean (0x76=failure, 0x89 = success)

         Changes: M_t

30   The *Write* command is used to update M_t according to the permissions in P_t. *The WR command by itself is not secure, since a clone QA Chip may simply return success every time.* Therefore a Write command should be followed by an authenticated read of M_t (e.g. via a Read command) to ensure that the change was actually made.

    The Write command is called by passing the WR command opcode followed by which M to be
35   updated, the new data to be written to M, and a digital signature of M. The data is sent least significant byte to most significant byte.

The ability to write to a specific 32-bit word within M_t is governed by the corresponding Permissions bits as stored in P_t. P_t can be set using the SetPermissions command.

The fact that M_t is Flash memory must be taken into account when writing the new value to M. It is possible for an attacker to remove power at any time. In addition, only the changes to M should be stored for maximum utilization. In addition, the longevity of M will need to be taken into account. This may result in the location of M being updated.

The signature is not keyed, since it must be generated by the consumable user.

The Write command is implemented with the following steps:

```
Loop through all of Flash, reading each word (will trigger checks)
Accept t
Restrict t to T
Accept M_E       # new M
Accept SIG_E


SIG_L = Generate SHA1[M_E]
If (SIG_L = SIG_E)
    output 0x76 # failure due to invalid signature
    exit
EndIf
DecEncountered ← 0
EqEncountered ← 0
For i ← msw to lsw #(word 15 to 0)
    P ← P_t[i]
    LT ← (M_E[i] < M_t[i])  # comparison is unsigned
    EQ ← (M_E[i] = M_t[i])
    WE ← (P = RW) ∨ ((P = MSR) ∧ LT) ∨ ((P = NMSR) ∧
(DecEncountered ∨ LT))
    DecEncountered ← ((P = MSR) ∧ LT)
                ∨ ((P = NMSR) ∧ DecEncountered)
                ∨ ((P = NMSR) ∧ EqEncountered ∧ LT)
    EqEncountered ← ((P = MSR) ∧ EQ) ∨ ((P = NMSR) ∧ EqEncountered
∧ EQ)


    If (¬WE) ∧ (M_E[i] ≠ M_t[i])
        output 0x76 # failure due to wanting a change but not allowed
it
```

727

```
        EndIf
    EndFor


    # At this point, M_E (desired) is correct to be written to the
    flash

    M_t ← M_E            # update flash
    output 0x89          # success
```

15.1.15 WriteAuth

> Input: n, R_E, M_E, SIG_E = [1 byte, 20 bytes, 64 bytes, 20 bytes]
>
> Output: Boolean ($0x76$=failure, $0x89$ = success)
>
> Changes:M_0, R_L

The *WriteAuth* command is used to securely replace the entire contents of M_0 (containing QA Chip application specific data) according to the P_{T+n}. The WriteAuth command only attempts to replace M_0 if the new value is signed combined with our local R.

It is only possible to sign messages by knowing K_n. This can be achieved by a call to the SignM command (because only a ChipS can know K_n). It means that without a chip that can be used to produce the required signature, a write of any value to M_0 is not possible.

The process is very similar to Write, except that if the validation succeeds, the M_E input parameter is processed against M_0 using permissions P_{T+n}.

The WriteAuth command is implemented with the following steps:

```
    Wait for MinTicksRemaining to become 0
    Loop through all of Flash, reading each word (will trigger checks)


    Accept n
    Restrict n to N
    Accept R_E
    Accept M_E
    SIG_L ← S_Kn[R_L|R_E|C_1|M_E] # calculation must take constant time
    Accept SIG_E
    If (SIG_E = SIG_L)
        Update R_L
        DecEncountered ← 0
        EqEncountered ← 0
        For i ← msw to lsw #(word 15 to 0)
            P ← P_T+n[i]
            LT ← (M_E[i] < M_0[i])  # comparison is unsigned
            EQ ← (M_E[i] = M_0[i])
```

```
WE  ←  (P  =  RW)  ∨  ((P  =  MSR)  ∧  LT)  ∨  ((P  =  NMSR)  ∧
(DecEncountered ∨ LT))
        DecEncountered ← ((P = MSR) ∧ LT)
                ∨ ((P = NMSR) ∧ DecEncountered)
                ∨ ((P = NMSR) ∧ EqEncountered ∧ LT)
        EqEncountered  ←  ((P  =  MSR)  ∧  EQ)  ∨  ((P  =  NMSR)  ∧
EqEncountered ∧ EQ)
        If ((¬WE) ∧ (M_E[i] ≠ M_0[i]))
            output 0x76 #  failure  due  to  wanting  a  change  but  not
allowed it
        EndIf
    EndFor
    # At this point, M_E (desired) is correct to be written to the
flash
    M_0 ← M_E          # update flash
    output 0x89            # success
EndIf
MinTicksRemaining ← MinTicks
```

16   Manufacture

This chapter makes some general comments about the manufacture and implementation of authentication chips. While the comments presented here are general, see [84] for a detailed description of an implementation of an authentication chip.

The authentication chip algorithms do not constitute a strong encryption device. The net effect is that they can be safely manufactured in any country (including the USA) and exported to anywhere in the world.

The circuitry of the authentication chip must be resistant to physical attack. A summary of manufacturing implementation guidelines is presented, followed by specification of the chip's physical defenses (ordered by attack).

Note that manufacturing comments are in addition to any legal protection undertaken, such as patents, copyright, and license agreements (for example, penalties if caught reverse engineering the authentication chip).

16.1   GUIDELINES FOR MANUFACTURING

The following are general guidelines for implementation of an authentication chip in terms of manufacture (see [84] for a detailed description of an authentication chip). *No special security is required during the manufacturing process.*

- Standard process

- Minimum size (if possible)
- Clock Filter
- Noise Generator
- Tamper Prevention and Detection circuitry
5 - Protected memory with tamper detection
- Boot circuitry for loading program code
- Special implementation of FETs for key data paths
- Data connections in polysilicon layers where possible
- OverUnderPower Detection Unit
10 - No test circuitry
- Transparent epoxy packaging

Finally, as a general note to manufacturers of Systems, the data line to the System authentication chip and the data line to the Consumable authentication chip must not be the same line. See Section 16.2.3 on page 736.

15 16.1.1   Standard Process

The authentication chip should be implemented with a standard manufacturing process (such as Flash). This is necessary to:

- allow a great range of manufacturing location options
- take advantage of well-defined and well-behaved technology
20 - reduce cost

Note that the standard process still allows physical protection mechanisms.

16.1.2   Minimum size

The authentication chip must have a low manufacturing cost in order to be included as the authentication mechanism for low cost consumables. It is therefore desirable to keep the chip size
25 as low as reasonably possible.

Each authentication chip requires 962 bits of non-volatile memory. In addition, the storage required for optimized HMAC-SHA1 is 1024 bits. The remainder of the chip (state machine, processor, CPU or whatever is chosen to implement Protocol C1) must be kept to a minimum in order that the number of transistors is minimized and thus the cost per chip is minimized. The circuit areas that
30 process the secret key information or could reveal information about the key should also be minimized (see Section 16.1.8 on page 734 for special data paths).

16.1.3   Clock Filter

The authentication chip circuitry is designed to operate within a specific clock speed range. Since the user directly supplies the clock signal, it is possible for an attacker to attempt to introduce race-
35 conditions in the circuitry at specific times during processing. An example of this is where a high clock speed (higher than the circuitry is designed for) may prevent an XOR from working properly,

and of the two inputs, the first may always be returned. These styles of transient fault attacks can be very efficient at recovering secret key information, and have been documented in [5] and [1]. The lesson to be learned from this is that the input clock signal *cannot be trusted*.

Since the input clock signal cannot be trusted, it must be limited to operate up to a maximum frequency. This can be achieved a number of ways.

One way to filter the clock signal is to use an edge detect unit passing the edge on to a delay, which in turn enables the input clock signal to pass through.

Figure 348 shows clock signal flow within the Clock Filter.

The delay should be set so that the maximum clock speed is a particular frequency (e.g. about 4 MHz). Note that this delay is not programmable - it is fixed.

The filtered clock signal would be further divided internally as required.

16.1.4   Noise Generator

Each authentication chip should contain a noise generator that generates continuous circuit noise. The noise will interfere with other electromagnetic emissions from the chip's regular activities and add noise to the I_{dd} signal. Placement of the noise generator is not an issue on an authentication chip due to the length of the emission wavelengths.

The noise generator is used to generate electronic noise, multiple state changes each clock cycle, and as a source of pseudo-random bits for the Tamper Prevention and Detection circuitry (see Section 16.1.5 on page 731).

A simple implementation of a noise generator is a 64-bit maximal period LFSR seeded with a non-zero number. The clock used for the noise generator should be running at the maximum clock rate for the chip in order to generate as much noise as possible.

16.1.5   Tamper Prevention and Detection circuitry

A set of circuits is required to test for and prevent physical attacks on the authentication chip. However what is actually detected as an attack may not be an intentional physical attack. It is therefore important to distinguish between these two types of attacks in an authentication chip:

•   where you *can be certain* that a physical attack has occurred.

•   where you *cannot* be certain that a physical attack has occurred.

The two types of detection differ in what is performed as a result of the detection. In the first case, where the circuitry can be certain that a true physical attack has occurred, erasure of Flash memory key information is a sensible action. In the second case, where the circuitry cannot be sure if an attack has occurred, there is still certainly something wrong. Action must be taken, but the action should not be the erasure of secret key information. A suitable action to take in the second case is a chip RESET. If what was detected was an attack that has permanently damaged the chip, the same conditions will occur next time and the chip will RESET again. If, on the other hand, what was detected was part of the normal operating environment of the chip, a RESET will not harm the key.

A good example of an event that circuitry cannot have knowledge about, is a power glitch. The glitch may be an intentional attack, attempting to reveal information about the key. It may, however, be the result of a faulty connection, or simply the start of a power-down sequence. It is therefore best to only RESET the chip, and not erase the key. If the chip was powering down, nothing is lost.

5   If the System is faulty, repeated RESETs will cause the consumer to get the System repaired. In both cases the consumable is still intact.

A good example of an event that circuitry can have knowledge about, is the cutting of a data line within the chip. If this attack is somehow detected, it could only be a result of a faulty chip (manufacturing defect) or an attack. In either case, the erasure of the secret information is a

10   sensible step to take.

Consequently each authentication chip should have 2 Tamper Detection Lines - one for definite attacks, and one for possible attacks. Connected to these Tamper Detection Lines would be a number of Tamper Detection test units, each testing for different forms of tampering. *In addition, we want to ensure that the Tamper Detection Lines and Circuits themselves cannot also be tampered*

15   *with.*

At one end of the Tamper Detection Line is a source of pseudo-random bits (clocking at high speed compared to the general operating circuitry). The Noise Generator circuit described above is an adequate source. The generated bits pass through two different paths - one carries the original data, and the other carries the inverse of the data. The wires carrying these bits are in the layer

20   above the general chip circuitry (for example, the memory, the key manipulation circuitry etc.). The wires must also cover the random bit generator. The bits are recombined at a number of places via an XOR gate. If the bits are different (they should be), a 1 is output, and used by the particular unit (for example, each output bit from a memory read should be ANDed with this bit value). The lines finally come together at the Flash memory Erase circuit, where a complete erasure is triggered by a

25   0 from the XOR. Attached to the line is a number of triggers, each detecting a physical attack on the chip. Each trigger has an oversize nMOS transistor attached to GND. The Tamper Detection Line physically goes through this nMOS transistor. If the test fails, the trigger causes the Tamper Detect Line to become 0. The XOR test will therefore fail on either this clock cycle or the next one (on average), thus RESETing or erasing the chip.

30   Figure 349 illustrates the basic principle of a Tamper Detection Line in terms of tests and the XOR connected to either the Erase or RESET circuitry.

The Tamper Detection Line must go through the drain of an output transistor for each test, as illustrated by Figure 350:

It is not possible to break the Tamper Detect Line since this would stop the flow of 1s and 0s from

35   the random source. The XOR tests would therefore fail. As the Tamper Detect Line physically passes through each test, it is not possible to eliminate any particular test without breaking the Tamper Detect Line.

It is important that the XORs take values from a variety of places along the Tamper Detect Lines in order to reduce the chances of an attack. Figure 351 illustrates the taking of multiple XORs from the Tamper Detect Line to be used in the different parts of the chip. Each of these XORs can be considered to be generating a ChipOK bit that can be used within each unit or sub-unit.

5 A sample usage would be to have an OK bit in each unit that is ANDed with a given ChipOK bit each cycle. The OK bit is loaded with 1 on a RESET. If OK is 0, that unit will fail until the next RESET. If the Tamper Detect Line is functioning correctly, the chip will either RESET or erase all key information. If the RESET or erase circuitry has been destroyed, then this unit will not function, thus thwarting an attacker.

10 The destination of the RESET and Erase line and associated circuitry is very context sensitive. It needs to be protected in much the same way as the individual tamper tests. There is no point generating a RESET pulse if the attacker can simply cut the wire leading to the RESET circuitry. The actual implementation will depend very much on what is to be cleared at RESET, and how those items are cleared.

15 Finally, Figure 352 shows how the Tamper Lines cover the noise generator circuitry of the chip. The generator and NOT gate are on one level, while the Tamper Detect Lines run on a level above the generator.

16.1.6   Protected memory with tamper detection

It is not enough to simply store secret information or program code in Flash memory. The Flash
20 memory and RAM must be protected from an attacker who would attempt to modify (or set) a particular bit of program code or key information. The mechanism used must conform to being used in the Tamper Detection Circuitry (described above).

The first part of the solution is to ensure that the Tamper Detection Line passes directly above each Flash or RAM bit. This ensures that an attacker cannot probe the contents of Flash or RAM. A
25 breach of the covering wire is a break in the Tamper Detection Line. The breach causes the Erase signal to be set, thus deleting any contents of the memory. The high frequency noise on the Tamper Detection Line also obscures passive observation.

The second part of the solution for Flash is to use multi-level data storage, but only to use a subset of those multiple levels for valid bit representations. Normally, when multi-level Flash storage is
30 used, a single floating gate holds more than one bit. For example, a 4-voltage-state transistor can represent two bits. Assuming a minimum and maximum voltage representing 00 and 11 respectively, the two middle voltages represent 01 and 10. In the authentication chip, we can use the two middle voltages to represent a single bit, and consider the two extremes to be invalid states. If an attacker attempts to force the state of a bit one way or the other by closing or cutting
35 the gate's circuit, an invalid voltage (and hence invalid state) results.

The second part of the solution for RAM is to use a parity bit. The data part of the register can be checked against the parity bit (which will not match after an attack).

733

The bits coming from Flash and RAM can therefore be validated by a number of test units (one per bit) connected to the common Tamper Detection Line. The Tamper Detection circuitry would be the first circuitry the data passes through (thus stopping an attacker from cutting the data lines).

While the multi-level Flash protection is enough for non-secret information, such as program code, R, and MinTicks, it is not sufficient for protecting K_1 and K_2. If an attacker adds electrons to a gate (see Section 5.7.2.15 on page 656) representing a single bit of K_1, and the chip boots up yet doesn't activate the Tamper Detection Line, the key bit must have been a 0. If it does activate the Tamper Detection Line, it must have been a 1. For this reason, all other non-volatile memory can activate the Tamper Detection Line, but K_1 and K_2 must not. Consequently Checksum is used to check for tampering of K_1 and K_2. A signature of the expanded form of K_1 and K_2 (i.e. 320 bits instead of 160 bits for each of K_1 and K_2) is produced, and the result compared against the Checksum. Any non-match causes a clear of all key information.

16.1.7   Boot circuitry for loading program code

Program code should be kept in multi-level Flash instead of ROM, since ROM is subject to being altered in a non-testable way. A boot mechanism is therefore required to load the program code into Flash memory (Flash memory is in an indeterminate state after manufacture).

The boot circuitry must not be in ROM - a small state-machine would suffice. Otherwise the boot code could be modified in an undetectable way.

The boot circuitry must erase all Flash memory, check to ensure the erasure worked, and then load the program code. Flash memory must be erased before loading the program code. Otherwise an attacker could put the chip into the boot state, and then load program code that simply extracted the existing keys. The state machine must also check to ensure that all Flash memory has been cleared (to ensure that an attacker has not cut the Erase line) before loading the new program code.

The loading of program code must be undertaken by the secure Programming Station before secret information (such as keys) can be loaded. This step must be undertaken as the first part of the programming process.

16.1.8   Special implementation of FETs for key data paths

The normal situation for FET implementation for the case of a CMOS Inverter (which involves a pMOS transistor combined with an nMOS transistor) as shown in Figure 353:

During the transition, there is a small period of time where both the nMOS transistor and the pMOS transistor have an intermediate resistance. The resultant power-ground short circuit causes a temporary increase in the current, and in fact accounts for the majority of current consumed by a CMOS device. A small amount of infrared light is emitted during the short circuit, and can be viewed through the silicon substrate (silicon is transparent to infrared light). A small amount of light is also emitted during the charging and discharging of the transistor gate capacitance and transmission line capacitance.

734

For circuitry that manipulates secret key information, such information must be kept hidden. An alternative non-flashing CMOS implementation should therefore be used for all data paths that manipulate the key or a partially calculated value that is based on the key.

The use of two non-overlapping clocks $\phi1$ and $\phi2$ can provide a non-flashing mechanism. $\phi1$ is connected to a second gate of all nMOS transistors, and $\phi2$ is connected to a second gate of all pMOS transistors. The transition can only take place in combination with the clock. Since $\phi1$ and $\phi2$ are non-overlapping, the pMOS and nMOS transistors will not have a simultaneous intermediate resistance. The setup is shown in Figure 354:

Finally, regular CMOS inverters can be positioned near critical non-Flashing CMOS components. These inverters should take their input signal from the Tamper Detection Line above. Since the Tamper Detection Line operates multiple times faster than the regular operating circuitry, the net effect will be a high rate of light-bursts next to each non-Flashing CMOS component. Since a bright light overwhelms observation of a nearby faint light, an observer will not be able to detect what switching operations are occurring in the chip proper. These regular CMOS inverters will also effectively increase the amount of circuit noise, reducing the SNR and obscuring useful EMI.

There are a number of side effects due to the use of non-Flashing CMOS:

- The effective speed of the chip is reduced by twice the rise time of the clock per clock cycle. This is not a problem for an authentication chip.
- The amount of current drawn by the non-Flashing CMOS is reduced (since the short circuits do not occur). However, this is offset by the use of regular CMOS inverters.
- Routing of the clocks increases chip area, especially since multiple versions of $\phi1$ and $\phi2$ are required to cater for different levels of propagation. The estimation of chip area is double that of a regular implementation.
- Design of the non-Flashing areas of the authentication chip are slightly more complex than to do the same with a with a regular CMOS design. In particular, standard cell components cannot be used, making these areas full custom. This is not a problem for something as small as an authentication chip, particularly when the entire chip does not have to be protected in this manner.

16.1.9   Connections in polysilicon layers where possible

Wherever possible, the connections along which the key or secret data flows, should be made in the polysilicon layers. Where necessary, they can be in metal 1, but must never be in the top metal layer (containing the Tamper Detection Lines).

16.1.10   OverUnderPower Detection Unit

Each authentication chip requires an OverUnderPower Detection Unit to prevent Power Supply Attacks. An OverUnderPower Detection Unit detects power glitches and tests the power level against a Voltage Reference to ensure it is within a certain tolerance. The Unit contains a single

Voltage Reference and two comparators. The OverUnderPower Detection Unit would be connected into the RESET Tamper Detection Line, thus causing a RESET when triggered.

A side effect of the OverUnderPower Detection Unit is that as the voltage drops during a power-down, a RESET is triggered, thus erasing any work registers.

5   16.1.11   No test circuitry

Test hardware on an authentication chip could very easily introduce vulnerabilities. As a result, the authentication chip should not contain any BIST or scan paths.

The authentication chip *must therefore be testable with external test vectors*. This should be possible since the authentication chip is not complex.

10   16.1.12   Transparent epoxy packaging

The authentication chip needs to be packaged in transparent epoxy so it can be photo-imaged by the programming station to prevent Trojan horse attacks. The transparent packaging does not compromise the security of the authentication chip since an attacker can fairly easily remove a chip from its packaging. For more information see Section 16.2.20 on page 743 and [86].

15   16.2   RESISTANCE TO PHYSICAL ATTACKS

While this chapter only describes manufacture in general terms (since this document does not cover a specific implementation of a Protocol C1 authentication chip), we can still make some observations about such a chip's resistance to physical attack. A description of the general form of each physical attack can be found in Section 5.7.2 on page 652.

20   16.2.1   Reading ROM

This attack depends on the key being stored in an addressable ROM. Since each authentication chip stores its authentication keys in internal Flash memory and not in an addressable ROM, this attack is irrelevant.

16.2.2   Reverse engineering the chip

25   Reverse engineering a chip is only useful when the security of authentication lies in the algorithm alone. However our authentication chips rely on a secret key, and not in the secrecy of the algorithm. Our authentication algorithm is, by contrast, public, and in any case, an attacker of a high volume consumable is assumed to have been able to obtain detailed plans of the internals of the chip.

30   In light of these factors, reverse engineering the chip itself, as opposed to the stored data, poses no threat.

16.2.3   Usurping the authentication process

There are several forms this attack can take, each with varying degrees of success. In all cases, it is assumed that a clone manufacturer will have access to both the System and the consumable

35   designs.

An attacker may attempt to build a chip that tricks the System into returning a valid code instead of generating an authentication code. This attack is not possible for two reasons. The first reason is

that System authentication chips and Consumable authentication chips, although physically identical, are programmed differently. In particular, the RD opcode and the RND opcode are the same, as are the WR and TST opcodes. A System authentication Chip cannot perform a RD command since every call is interpreted as a call to RND instead. The second reason this attack

5 would fail is that separate serial data lines are provided from the System to the System and Consumable authentication chips. Consequently neither chip can see what is being transmitted to or received from the other.

If the attacker builds a clone chip that ignores WR commands (which decrement the consumable remaining), Protocol C1 ensures that the subsequent RD will detect that the WR did not occur. The

10 System will therefore not go ahead with the use of the consumable, thus thwarting the attacker. The same is true if an attacker simulates loss of contact before authentication - since the authentication does not take place, the use of the consumable doesn't occur.

An attacker is therefore limited to modifying each System in order for clone consumables to be accepted (see Section 16.2.4 on page 737 for details of resistance this attack).

15 16.2.4   Modification of system

The simplest method of modification is to replace the System's authentication chip with one that simply reports success for each call to TST. This can be thwarted by System calling TST several times for each authentication, with the first few times providing false values, and expecting a fail from TST. The final call to TST would be expected to succeed. The number of false calls to TST

20 could be determined by some part of the returned result from RD or from the system clock. Unfortunately an attacker could simply rewire System so that the new System clone authentication chip can monitor the returned result from the consumable chip or clock. The clone System authentication chip would only return success when that monitored value is presented to its TST function. Clone consumables could then return any value as the hash result for RD, as the clone

25 System chip would declare that value valid. There is therefore no point for the System to call the System authentication chip multiple times, since a rewiring attack will only work for the System that has been rewired, and not for all Systems.

A similar form of attack on a System is a replacement of the System ROM. The ROM program code can be altered so that the Authentication never occurs. There is nothing that can be done about

30 this, since the System remains in the hands of a consumer. Of course this would void any warranty, but the consumer may consider the alteration worthwhile if the clone consumable were extremely cheap and more readily available than the original item.

The System/consumable manufacturer must therefore determine how likely an attack of this nature is. Such a study must include given the pricing structure of Systems and Consumables, frequency

35 of System service, advantage to the consumer of having a physical modification performed, and where consumers would go to get the modification performed.

The likelihood of physical alteration increases with the perceived artificiality of the consumable marketing scheme. It is one thing for a consumable to be protected against clone manufacturers. It is quite another for a consumable's market to be protected by a form of exclusive licensing arrangement that creates what is viewed by consumers as artificial markets. In the former case, owners are not so likely to go to the trouble of modifying their system to allow a clone manufacturer's goods. In the latter case, consumers are far more likely to modify their System. A case in point is DVD. Each DVD is marked with a region code, and will only play in a DVD player from that region. Thus a DVD from the USA will not play in an Australian player, and a DVD from Japan, Europe or Australia will not play in a USA DVD player. Given that certain DVD titles are not available in all regions, or because of quality differences, pricing differences or timing of releases, many consumers have had their DVD players modified to accept DVDs from *any* region. The modification is usually simple (it often involves soldering a single wire), voids the owner's warranty, and often costs the owner some money. But the interesting thing to note is that the change is not made so the consumer can use clone consumables - the consumer will still only buy real consumables, but from different regions. The modification is performed to remove what is viewed as an <u>artificial</u> barrier, placed on the consumer by the movie companies. In the same way, a System/Consumable scheme that is viewed as unfair will result in people making modifications to their Systems.

The limit case of modifying a system is for a clone manufacturer to provide a completely clone System which takes clone consumables. This may be simple competition or violation of patents. Either way, it is beyond the scope of the authentication chip and depends on the technology or service being cloned.

16.2.5   Direct viewing of chip operation by conventional probing

In order to view the chip operation, the chip must be operating. However, the Tamper Prevention and Detection circuitry covers those sections of the chip that process or hold the key. It is not possible to view those sections through the Tamper Prevention lines.

An attacker cannot simply slice the chip past the Tamper Prevention layer, for this will break the Tamper Detection Lines and cause an erasure of all keys at power-up. Simply destroying the erasure circuitry is not sufficient, since the multiple ChipOK bits (now all 0) feeding into multiple units within the authentication chip will cause the chip's regular operating circuitry to stop functioning.

To set up the chip for an attack, then, requires the attacker to delete the Tamper Detection lines, stop the Erasure of Flash memory, and somehow rewire the components that relied on the ChipOK lines. Even if all this could be done, the act of slicing the chip to this level will most likely destroy the charge patterns in the non-volatile memory that holds the keys, making the process fruitless.

16.2.6   Direct viewing of the non-volatile memory

If the authentication chip were sliced so that the floating gates of the Flash memory were exposed, without discharging them, then the keys could probably be viewed directly using an STM or SKM. However, slicing the chip to this level without discharging the gates is probably impossible. Using wet etching, plasma etching, ion milling, or chemical mechanical polishing will almost certainly

5 discharge the small charges present on the floating gates. This is true of regular Flash memory, but even more so of multi-level Flash memory.

16.2.7   Viewing the light bursts caused by state changes

All sections of circuitry that manipulate secret key information are implemented in the non-Flashing CMOS described above. This prevents the emission of the majority of light bursts. Regular CMOS

10 inverters placed in close proximity to the non-Flashing CMOS will hide any faint emissions caused by capacitor charge and discharge. The inverters are connected to the Tamper Detection circuitry, so they change state many times (at the high clock rate) for each non-Flashing CMOS state change.

16.2.8   Viewing the keys using an SEPM

15 An SEPM attack can be simply thwarted by adding a metal layer to cover the circuitry. However an attacker could etch a hole in the layer, so this is not an appropriate defense.

The Tamper Detection circuitry described above will shield the signal as well as cause circuit noise. The noise will actually be a greater signal than the one that the attacker is looking for. If the attacker attempts to etch a hole in the noise circuitry covering the protected areas, the chip will not function,

20 and the SEPM will not be able to read any data.

An SEPM attack is therefore fruitless.

16.2.9   Monitoring EMI

The Noise Generator described above will cause circuit noise. The noise will interfere with other electromagnetic emissions from the chip's regular activities and thus obscure any meaningful

25 reading of internal data transfers.

16.2.10   Viewing I_{dd} fluctuations

The solution against this kind of attack is to decrease the SNR in the I_{dd} signal. This is accomplished by increasing the amount of circuit noise and decreasing the amount of signal.

The Noise Generator circuit (which also acts as a defense against EMI attacks) will also cause

30 enough state changes each cycle to obscure any meaningful information in the I_{dd} signal.

In addition, the special Non-Flashing CMOS implementation of the key-carrying data paths of the chip prevents current from flowing when state changes occur. This has the benefit of reducing the amount of signal.

16.2.11   Differential fault analysis

35 Differential fault bit errors are introduced in a non-targeted fashion by ionization, microwave radiation, and environmental stress. The most likely effect of an attack of this nature is a change in

Flash memory (causing an invalid state) or RAM (bad parity). Invalid states and bad parity are detected by the Tamper Detection Circuitry, and cause an erasure of the key.

Since the Tamper Detection Lines cover the key manipulation circuitry, any error introduced in the key manipulation circuitry will be mirrored by an error in a Tamper Detection Line. If the Tamper Detection Line is affected, the chip will either continually RESET or simply erase the key upon a power-up, rendering the attack fruitless.

Rather than relying on a non-targeted attack and hoping that "just the right part of the chip is affected in just the right way", an attacker is better off trying to introduce a targeted fault (such as overwrite attacks, gate destruction etc.). For information on these targeted fault attacks, see the relevant sections below.

16.2.12   Clock glitch attacks

The Clock Filter (described above) eliminates the possibility of clock glitch attacks.

16.2.13   Power supply attacks

The OverUnderPower Detection Unit (described above) eliminates the possibility of power supply attacks.

16.2.14   Overwriting ROM

Authentication chips store program code, keys and secret information in Flash memory, and not in ROM. This attack is therefore not possible.

16.2.15   Modifying EEPROM/Flash

Authentication chips store program code, keys and secret information in multi-level Flash memory. However the Flash memory is covered by two Tamper Prevention and Detection Lines. If either of these lines is broken (in the process of destroying a gate via a laser-cutter) the attack will be detected on power-up, and the chip will either RESET (continually) or erase the keys from Flash memory. This process is described in Section 16.1.6 on page 733.

Even if an attacker is able to somehow access the bits of Flash and destroy or short out the gate holding a particular bit, this will force the bit to have no charge or a full charge. These are both invalid states for the authentication chip's usage of the multi-level Flash memory (only the two middle states are valid). When that data value is transferred from Flash, detection circuitry will cause the Erasure Tamper Detection Line to be triggered - thereby erasing the remainder of Flash memory and RESETing the chip. This is true for program code, and non-secret information. As key data is read from multi-level flash memory, it is not imediately checked for validity (otherwise information about the key is given away). Instead, a specific key validation mechanism is used to protect the secret key information.

An attacker could theoretically etch off the upper levels of the chip, and deposit enough electrons to change the state of the multi-level Flash memory by 1/3. If the beam is high enough energy it might be possible to focus the electron beam through the Tamper Prevention and Detection Lines. As a result, the authentication chip must perform a validation of the keys before replying to the Random,

Test or Random commands. The SHA-1 algorithm must be run on the keys, and the results compared against an internal checksum value. This gives an attacker a 1 in 2^{160} chance of tricking the chip, which is the same chance as guessing either of the keys.

A Modify EEPROM/Flash attack is therefore fruitless.

16.2.16 Gate destruction attacks

Gate Destruction Attacks rely on the ability of an attacker to modify a single gate to cause the chip to reveal information during operation. However any circuitry that manipulates secret information is covered by one of the two Tamper Prevention and Detection lines. If either of these lines is broken (in the process of destroying a gate) the attack will be detected on power-up, and the chip will either RESET (continually) or erase the keys from Flash memory.

To launch this kind of attack, an attacker must first reverse-engineer the chip to determine which gate(s) should be targeted. Once the location of the target gates has been determined, the attacker must break the covering Tamper Detection line, stop the Erasure of Flash memory, and somehow rewire the components that rely on the ChipOK lines. Rewiring the circuitry cannot be done without slicing the chip, and even if it could be done, the act of slicing the chip to this level will most likely destroy the charge patterns in the non-volatile memory that holds the keys, making the process fruitless.

16.2.17 Overwrite attack

An overwrite attack relies on being able to set individual bits of the key without knowing the previous value. It relies on probing the chip, as in the conventional probing attack and destroying gates as in the gate destruction attack. Both of these attacks (as explained in their respective sections), will not succeed due to the use of the Tamper Prevention and Detection Circuitry and ChipOK lines.

However, even if the attacker is able to somehow access the bits of Flash and destroy or short out the gate holding a particular bit, this will force the bit to have no charge or a full charge. These are both invalid states for the authentication chip's usage of the multi-level Flash memory (only the two middle states are valid). When that data value is transferred from Flash detection circuitry will cause the Erasure Tamper Detection Line to be triggered - thereby erasing the remainder of Flash memory and RESETing the chip. In the same way, a parity check on tampered values read from RAM will cause the Erasure Tamper Detection Line to be triggered.

An overwrite attack is therefore fruitless.

16.2.18 Memory remanence attack

Any working registers or RAM within the authentication chip may be holding part of the authentication keys when power is removed. The working registers and RAM would continue to hold the information for some time after the removal of power. If the chip were sliced so that the

gates of the registers/RAM were exposed, without discharging them, then the data could probably be viewed directly using an STM.

5   The first defense can be found above, in the description of defense against power glitch attacks. When power is removed, all registers and RAM are cleared, just as the RESET condition causes a clearing of memory.

10   The chances then, are less for this attack to succeed than for a reading of the Flash memory. RAM charges (by nature) are more easily lost than Flash memory. The slicing of the chip to reveal the RAM will certainly cause the charges to be lost (if they haven't been lost simply due to the memory not being refreshed and the time taken to perform the slicing).

This attack is therefore fruitless.

16.2.19 Chip theft attack

There are distinct phases in the lifetime of an authentication chip. Chips can be stolen when at any of these stages:

- After manufacture, but before programming of key
- After programming of key, but before programming of state data
- After programming of state data, but before insertion into the consumable or system
- After insertion into the system or consumable

A theft in between the chip manufacturer and programming station would only provide the clone manufacturer with blank chips. This merely compromises the sale of authentication chips, not anything authenticated by the authentication chips. Since the programming station is the only mechanism with consumable and system product keys, a clone manufacturer would not be able to program the chips with the correct key. Clone manufacturers would be able to program the blank chips for their own Systems and Consumables, but it would be difficult to place these items on the market without detection.

The second form of theft can only happen in a situation where an authentication chip passes through two or more distinct programming phases. This is possible, but unlikely. In any case, the worst situation is where no state data has been programmed, so all of M is read/write. If this were the case, an attacker could attempt to launch an adaptive chosen text attack on the chip. The HMAC-SHA1 algorithm is resistant to such attacks. For more information see Section 14.7 on page 699.

The third form of theft would have to take place in between the programming station and the installation factory. The authentication chips would already be programmed for use in a particular system or for use in a particular consumable. The only use these chips have to a thief is to place them into a clone System or clone Consumable. Clone systems are irrelevant - a cloned System would not even require an authentication chip. For clone Consumables, such a theft would limit the number of cloned products to the number of chips stolen. A single theft should not create a supply constant enough to provide clone manufacturers with a cost-effective business.

The final form of theft is where the System or Consumable itself is stolen. When the theft occurs at the manufacturer, physical security protocols must be enhanced. If the theft occurs anywhere else, it is a matter of concern only for the owner of the item and the police or insurance company. The security mechanisms that the authentication chip uses assume that the consumables and systems are in the hands of the public. Consequently, having them stolen makes no difference to the security of the keys.

16.2.20 Trojan horse attack

A Trojan horse attack involves an attacker inserting a fake authentication chip into the programming station and retrieving the same chip after it has been programmed with the secret key information.

The difficulty of these two tasks depends on both logical and physical security, but is an expensive attack - the attacker has to manufacture a false authentication chip, and it will only be useful where the effort is worth the gain. For example, obtaining the secret key for a specific car's authentication chip is most likely not worth an attacker's efforts, while the key for a printer's ink cartridge may be

5   very valuable.

The problem arises if the programming station is unable to tell a Trojan horse authentication chip from a real one - which is the problem of authenticating the authentication chip.

One solution to the authentication problem is for the manufacturer to have a programming station attached to the end of the production line. Chips passing the manufacture QA tests are

10 programmed with the manufacturer's secret key information. The chip can therefore be verified by the C1 authentication protocol, and give information such as the expected batch number, serial number etc. The information can be verified and recorded, and the valid chip can then be reprogrammed with the System or Consumable key and state data. An attacker would have to substitute an authentication chip with a Trojan horse programmed with the manufacturer's secret

15 key information and copied batch number data from the removed authentication chip. This is only possible if the manufacturer's secret key is compromised (the key is changed regularly and not known by a human) or if the physical security at the manufacturing plant is compromised at the end of the manufacturing chain.

Even if the solution described were to be undertaken, the possibility of a Trojan horse attack does

20 not go away - it merely is removed to the manufacturer's physical location. A better solution requires no physical security at the manufacturing location.

The preferred solution then, is to use transparent epoxy on the chip's packaging and to image the chip before programming it. Once the chip has been mounted for programming it is in a known fixed orientation. It can therefore be high resolution photo-imaged and X-rayed from multiple directions,

25 and the images compared against "signature" images. Any chip not matching the image signature is treated as a Trojan horse and rejected.

1 REFILL OF INK IN PRINTERS - Printer based refill device

1.1 FUNCTIONAL PURPOSE

The functional purpose of the printer based refill device is as follows:

- To refill ink into printers by physically connecting the refill device to the printer.
- To ensure that the correct ink is used for the correct operation of the printer (i.e. will not damage the printhead).
- To ensure accurate measure of ink is transferred from the refilling device to the printer during refills.
- The refill device is controlled by the printer. Apart from the QA Chip[1] the refill device has no other processing power.

1.2 BASIC COMPONENTS OF THE REFILL DEVICE

Figure 355 shows the components of the printer based refill device.

The printer based refill device will consist of following components:

- An ink reservoir - which stores the ink. Each refill device will allow ink reservoirs of various capacities. When the ink reservoir empties out, it is replaced by another reservoir containing more ink of the same type or different type or refilled (for example through a refill station as described in Section 2 and Section 3).
- An ink output device- which dispenses ink to the printer being refilled when physically connected to the printer.
- A QA Chip and associated circuitry - which stores the amount of ink in the reservoir along with the attributes of the ink in a digital format.
- The electrical connections to the QA Chip.
- NB - No additional microprocessors are required to be present in the refill device. Hence the refill device uses the processing power of the printer to oversee the refilling process.
- An ink transfer mechanism (optional) which controls the flow ink from the refill device to the printer and is controlled by the printer. Therefore the control connections for the ink transfer mechanism will be connected to the printer.
- Alternatively, the ink transfer mechanism could be in the printer. Refer to Section 1.3.

1.3 PRINTER DESCRIPTION AND FUNCTIONS

Printers which will be refilled by these refilling devices must have the following components:

- Microprocessor assembly which will control the refill procedure as described Section 1.4. The microprocessor assembly will access the QA Chip and ink transfer mechanism of the refill device.
- A QA Chip storing the ink amount remaining in the printer.

---

[1]**General Note**: Througout this document, if secure refilling is required then a physical QA Chip or any other virtual device performing the QA Chip protocol can be used. Refer to [1].

- An optional ink transfer mechanism to control the flow of ink from the refill device to the printer. This ink transfer mechanism must be present in the printer if the refill device doesn't have one of its own.

1.4 OPERATIONAL PROCEDURE

The operational procedure can be divided into two parts:

- Refilling printers using the refill device.
- Refilling of the ink reservoir in the refill device . See Section 2 and Section 3.

1.4.1 Refilling of printers

Figure 356 shows a printer being refilled by a printer based refill device. The ink transfer mechanism is located in the printer in this case. The ink transfer mechanism could be also located in the refill device as described in Section 1.2.

The following is a description for refilling of printers using the printer based refill device:

- Ink output device from the refilling device is connected to the printer.
- The QA Chip electrical connection is connected to the printer.
- The refill option is selected on the user interface of the printer. The microprocessor assembly in the printer will then do the following:

a.   Read ink attributes (for example ink type, ink characteristics, ink colour, ink manufacturer etc) stored in the QA Chip of the ink reservoir unit. Refer to[1].

b. Compare the ink attributes as required by the printer for correct operation. This may require reading of data from the QA Chip in the printer.

c. Only if Step b is successful, then do the following:

i. Determine the amount of ink to be transferred by any or all of the following means, ensuring that the reservoir has enough ink for the transfer:

- Fixed amount (e.g. based on a pre-programmed value or printer model).
- User-selectable amount.

ii. Decrement the amount of ink transferred from the QA Chip in the refill station and increment the QA Chip in the printer (which stores the amount of ink in the printer) with corresponding ink amount.

iii. Command the ink transfer mechanism to release the ink to the printer through the output device.

2 Home use refill station

2.1 FUNCTIONAL PURPOSE

The functional purpose of the commercial refill station is as follows:

- To refill ink into ink cartridges at home or in a small office.
- Single ink cartridge is filled at a time.
- To ensure that the correct ink present in the refill station is transferred to the correct ink cartridge.
- To ensure accurate measure of ink is transferred from the refilling station to the ink cartridge during refills.
- The refilling station provides the processing power required to perform refills of ink cartridges.

2.2 BASIC COMPONENTS

Figure 357 shows the components of a home refill station.

A home refill station will consist of one of the following ink refill units:

- A single reservoir ink refill unit suitable for black ink (or any other single colour).

- A multi reservoir ink refill unit suitable for coloured ink for example CMY (Cyan, Magenta, Yellow).

2.2.1 Ink reservoir unit

Figure 358 shows the components of a three-ink reservoir unit.

The ink reservoir unit will consist of the following:

- Multiple ink reservoirs or a single ink reservoir which stores ink. Each refill station will allow ink reservoirs of various capacities. When the ink reservoir empties out, it is replaced by another reservoir containing more ink of the same or different type or refilled (for example through a refill station as described in Section 3).

- A QA Chip and associated circuitry in each of the ink reservoirs - which stores the amount of ink in the reservoir along with the attributes of the ink.

- The electrical connections to each of the QA Chips.

2.2.2 Ink transfer unit

The ink reservoir unit will consist of the following:

- Ink output device from each ink reservoir.

- The output ink transfer mechanism controls the flow ink from the ink refill unit to the ink cartridge and is controlled by the microprocessor assembly.

- Final ink output devices to the cartridge interface assembly

2.2.3 Cartridge interface unit

This unit will provide the physical interface to the ink cartridges. Each ink cartridge interface unit will hold a single or multiple cartridges of particular physical dimension.

The cartridge interface unit can removed from the ink refill unit and replaced with another interface unit to cater for other physically different cartridges.

2.2.4 Microprocessor assembly

The controls connections for the ink transfer mechanism and the electrical connections of the QA Chip are connected to the microprocessor assembly. The microprocessor assembly oversees and controls the refill process.

The microprocessor assembly will communicate with a user interface to accept commands and provide responses for various refill operations.

2.3 INK CARTRIDGE DESCRIPTION

Ink cartridges which will be refilled in a home refill station must have a QA Chip storing the following components:

- Ink amount remaining.

- Ink attributes (for example - ink type, ink characteristics, ink colour, ink manufacturer).

2.4 OPERATIONAL PROCEDURE

The operational procedure can be divided into two parts:

- Refilling of ink cartridges using the home refill station.
- Refilling the ink reservoirs used in the refill station is discussed in Section 3.

2.5 REFILLING OF INK CARTRIDGES USING THE HOME REFILL STATION

Figure 359 shows the refill of ink cartridges in a home refill station.

The following is a description for refilling of ink cartridges in the home refill station:

- Load the ink cartridge into the cartridge interface unit of the ink refill unit. This will connect the QA Chip of the ink cartridge to the microprocessor assembly. It will also connect the ink output device of the ink refill unit to the ink cartridge.
- The model number of the ink cartridge is read from the QA Chip by the microprocessor assembly controlling the ink refill units.
- The microprocessor assembly will determine whether the ink refill unit is suitable for the ink cartridge model.
- The refill option is selected on the microprocessor assembly through the user interface. The microprocessor assembly will then do the following:

a. Read ink attributes (for example ink type, ink characteristics, ink colour, ink manufacturer etc) stored in the QA Chip of the ink cartridge. Refer to[1].

b. Compare the read ink attributes to the ink attribute list in the refill station.This may also require reading of the ink attributes stored in the QA Chip of the ink reservoirs in the refill unit.

c. Only if Step b is successful, then do the following:

i. Determine the amount of ink to be transferred by any or all of the following means, ensuring that the reservoir has enough ink for the transfer:

- Fixed amount (e.g. based on a pre-programmed value ,cartridge model or reservoir type).
- User-selectable amount.

ii. Check the ink reservoir in the ink refill unit has adequate amount of ink to refill the ink cartridge

iii. Decrement the amount of ink transferred from the QA Chip in the ink refill unit and increment the QA Chip in the ink cartridge with corresponding ink amount.

iv. If incrementing of the QA Chip with ink amount is successful then a command is sent to the ink transfer mechanism to release the ink to the ink cartridge through the output device.

3 Commercial refill station

3.1 FUNCTIONAL PURPOSE

The functional purpose of the commercial refill station is as follows:

- To refill ink into ink cartridges that are taken to the refill station for refilling.
- Multiple ink cartridges of different models can be refilled.
- To ensure that the correct ink present in the refill station is transferred to the ink cartridge.
- To ensure accurate measure of ink is transferred from the refilling station to the ink cartridge during refills.

- The refilling station provides all processing power required to perform refills of ink cartridges.

3.2 BASIC COMPONENTS OF THE REFILL STATION

Figure 360 shows the components of a commercial refill station.

A commercial refill station will consist of multiple ink refill units controlled by a single microprocessor assembly. Each ink refill unit can refill a single ink cartridge at a time.

Each ink refill unit will consist of the following sub units:

- Ink reservoir unit
- Switch unit
- Ink transfer unit
- Multiple cartridge interface unit

3.2.1

Ink reservoir unit

Figure 361 shows the components of a ink reservoir unit.

The ink reservoir unit will consist of the following:

- Multiple ink reservoirs - which stores ink. Each refill device will allow ink reservoirs of various capacities. When the ink reservoir empties out, it is replaced by another reservoir containing more ink of the same or different type or refilled. Refer to Section 3.5.

- A QA Chip and associated circuitry in each of the ink reservoirs - which stores the amount of ink in the reservoir along with the attributes of the ink in digital format.

- The electrical connections of each of the QA Chips are connected to the microprocessor assembly.

3.2.2 Switch unit

This unit will switch the inks selected from different ink reservoirs to the ink transfer unit to be dispensed into ink cartridges.

The switch unit will prevent mixing of any residual ink left in dispensing devices after each ink cartridge is refilled.

3.2.3 Ink transfer unit

The ink reservoir unit will consist of the following:

- Ink output device from each ink reservoir.

- An output ink transfer mechanism which controls the flow ink from the ink refill unit to the ink cartridge and is controlled by the microprocessor assembly.

- Final ink output devices to the multiple cartridge interface assembly

3.2.4 Multiple cartridge interface unit

This unit will provide the physical interface to the ink cartridges. Each ink cartridge interface will hold cartridges of different physical dimensions.

Each cartridge interface unit can provide an interface for about 20 physically different cartridges.

The cartridge interface unit can removed from the ink refill unit and replaced with another interface unit to cater for other physically different cartridges.

749

3.2.5 Microprocessor assembly with a user interface

The controls connections for the ink transfer mechanism and the electrical connections of the QA Chip are connected to the microprocessor assembly. The microprocessor assembly will oversee and control the refill process.

The microprocessor assembly will communicate with a user interface to accept commands and provide responses for various refill operations.

3.3 INK CARTRIDGE DESCRIPTION

Ink cartridges which will be refilled in a commercial refill station must have a QA Chip storing the following components:

- Ink amount remaining.
- Ink attributes (for example - ink type, ink characteristics, ink colour, ink manufacturer).

3.4 OPERATIONAL PROCEDURE

The operational procedure can be divided into two parts:

- Refilling of ink cartridges using the commercial refill station.
- Refilling the ink reservoirs used in the refill station is covered in Section 3.5.

3.4.1 Refilling ink cartridges using the commercial refill station

Figure 362 shows the refill of ink cartridges in a commercial refill station.

The following is a description for refilling of ink cartridges in the commercial refill station:

- Load the ink cartridge into the multiple cartridge interface unit of the ink refill unit. This will connect the QA Chip of the ink cartridge to the microprocessor assembly. It will also connect the ink output device of the ink refill unit to the ink cartridge.
- The model number of the ink cartridge automatically is read from the QA Chip by the microprocessor assembly controlling the ink refill units.
- The microprocessor assembly will determine whether the ink refill unit is suitable for the ink cartridge model.
- The refill option is selected on the microprocessor assembly through the user interface. The microprocessor assembly will then do the following:

   a. Read ink attributes (for example ink type, ink characteristics, ink colour, ink manufacturer etc) stored in the QA Chip of the ink cartridge. Refer to[1].

   b. Compare the read ink attributes to the ink attribute list in the refill station. This may also require reading of the ink attributes stored in the QA Chip of the ink reservoirs in the refill unit.

   c. Only if Step b is successful, then do the following:

   i. Determine the amount of ink to be transferred by any or all of the following means, ensuring that the reservoir has enough ink for the transfer:

   - Fixed amount (e.g. based on a pre-programmed value, cartridge model or reservoir type).
   - User-selectable amount.

ii. The microprocessor assembly will calculate the cost of ink amount and interrogate the user for a payment method -credit card or cash. If credit card option is selected it will request a credit card number to be selected and interface to a payment system to complete the transaction before proceeding further.

iii. Decrement the amount of ink transferred from the QA Chip in the ink refill unit and increment the QA

5   Chip in the ink cartridge with corresponding ink amount.

iv. If incrementing of the QA Chip with ink amount is successful then a command is sent to the ink transfer mechanism to release the ink to the ink cartridge through the output device.

3.5   REFILLING THE INK RESERVOIRS

The ink reservoirs of any ink refill device can be refilled recursively by the procedure described in Section

10   3.4.1, the only exception being the ink cartridge replaced by the ink reservoir.

3.6   COMMERCIAL REFILL STATION FOR A PRODUCTION ENVIRONMENT

This refill station resembles a commercial refill station but fills multiple ink cartridges of the same type at the same time. This will serve as a filling station for new cartridges in a production environment.

LOGICAL INTERFACE SPECIFICATION FOR PREFERRED FORM OF QA CHIP

1   Introduction

This document defines the *QA Chip Logical Interface* , which provides authenticated manipulation of specific printer and consumable parameters. The interface is described in terms of data structures and the functions that manipulate them, together with examples of use. While the descriptions and examples are targetted towards the printer application, they are equally applicable in other domains.

2   Scope

The document describes the QA Chip Logical Interface as follows:

- data structures and their uses (Section 5 to Section 9).

- functions, including inputs, outputs, signature formats, and a logical implementation sequence (Section 10 to Section 30).

- typical functional sequences of printers and consumables, using the functions and data structures of the interface (Section 31 to Section 32).

The QA Chip Logical Interface is a *logical* interface, and is therefore implementation independent. Although this document does not cover implementation details on particular platforms, expected implementations include:

- Software only

- Off-the-shelf cryptographic hardware.

- ASICs, such as SBR4320 [2] and SOPEC [3] for physical insertion into printers and ink cartridges

- Smart cards.

3   Nomenclature

3.1   SYMBOLS

The following symbolic nomenclature is used throughout this document:

Table 246. Summary of symbolic nomenclature

Symbol	Description
F[X]	Function F, taking a single parameter X
F[X, Y]	Function F, taking two parameters, X and Y
X | Y	X concatenated with Y
X ∧ Y	Bitwise X AND Y
X ∨ Y	Bitwise X OR Y (inclusive-OR)
X ⊕ Y	Bitwise X XOR Y (exclusive-OR)
¬X	Bitwise NOT X (complement)

X ← Y	X is assigned the value Y
X ← {Y, Z}	The domain of assignment inputs to X is Y and Z
X = Y	X is equal to Y
X ≠ Y	X is not equal to Y
⇓X	Decrement X by 1 (floor 0)
⇑X	Increment X by 1 (modulo register length)
Erase X	Erase Flash memory register X
SetBits[X, Y]	Set the bits of the Flash memory register X based on Y
Z ← ShiftRight[X, Y]	Shift register X right one bit position, taking input bit from Y and placing the output bit in Z
a.b	Data field or member function 'b' in object a.

3.2 PSEUDOCODE

3.2.1 Asynchronous

The following pseudocode:

```
var = expression
```

means the var signal or output is equal to the evaluation of the expression.

3.2.2 Synchronous

The following pseudocode:

```
var ← expression
```

means the var register is assigned the result of evaluating the expression during this cycle.

3.2.3 Expression

Expressions are defined using the nomenclature in Table 246 above. Therefore:

```
var = (a = b)
```

is interpreted as the var signal is 1 if a is equal to b, and 0 otherwise.

4 TERMS

4.1 QA Device and System

An instance of a QA Chip Logical Interface (on any platform) is a *QA Device*.

QA Devices cannot talk directly to each other. A *System* is a logical entity which has one or more QA Devices connected logically (or physically) to it, and calls the functions on the QA Devices. The system is considered secure and the program running on the system is considered to be trusted.

4.2 Types of QA Devices

4.2.1 Trusted QA Device

The *Trusted QA Device* forms an integral part of the system itself and resides within the trusted environment of the system. It enables the system to extend trust to external QA Device s. The Trusted QA Device is only trusted because the system itself is trusted.

4.2.2 External untrusted QA Device

The *External untrusted QA Device* is a QA Device that resides external to the trusted environment of the system and is therefore untrusted. The purpose of the QA Chip Logical Interface is to allow the external untrusted QA Devices to become effectively trusted. This is accomplished when a Trusted QA Device shares a secret key with the external untrusted QA Device, or with a Translation QA Device (see below).

In a printing application external untrusted QA Devices would typically be instances of SBR4320 implementations located in a consumable or the printer.

4.2.3 Translation QA Device

A *Translation QA Device* is used to translate signatures between QA Devices and extend effective trust when secret keys are not directly shared between QA Devices.

The Translation QA Device must share a secret key with the Trusted QA Device that allows the Translation QA Device to effectively become trusted by the Trusted QA Device and hence trusted by the system. The Translation QA Device shares a different secret key with another external untrusted QA Device (which may in fact be a Translation QA Device etc). Although the Trusted QA Device doesn't share (know) the key of the external untrusted QA Device, signatures generated by that untrusted device can be translated by the Translation QA Device into signatures based on the key that the Trusted QA Device *does* know, and thus extend trust to the otherwise untrusted external QA Device.

In a SoPEC-based printing application, the Printer QA Device acts as a Translation QA Device since it shares a secret key with the SoPEC, and a different secret key with the ink carridges.

4.2.4 Consumable QA Device

A *Consumable QA Device* is an external untrusted QA Device located in a consumable. It typically contains details about the consumable, including how much of the consumable remains.

In a printing application the consumable QA Device is typically found in an ink cartridge and is referred to as an *Ink QA Device*, or simply *Ink QA* since ink is the most common consumable for printing applications. However, other consumables in printing applications include media and impression counts, so consumable QA Device is more generic.

4.2.5 Printer QA Device

A *Printer QA Device* is an external untrusted device located in the printer. It contains details about the operating parameters for the printer, and is often referred to as a *Printer QA*.

4.2.6 Value Upgrader QA Device

A *Value Upgrader QA Device* contains the necessary functions to allow a system to write an initial value (e.g. an ink amount) into another QA Device, typically a consumable QA Device . It also allows a system to refill/replenish a value in a consumable QA Device after use.

Whenever a value upgrader QA Device increases the amount of value in another QA Device , the value in the value upgrader QA Device is correspondingly decreased. This means the value upgrader QA Device cannot create value - it can only pass on whatever value it itself has been issued with. Thus a value upgrader QA Device can itself be replenished or topped up by another value upgrader QA Device.

An example of a value upgrader is an *Ink Refill QA Device*, which is used to fill/refill ink amount in an Ink QA Device.

4.2.7 Parameter Upgrader QA Device

A *Parameter Upgrader QA Device* contains the necessary functions to allow a system to write an initial parameter value (e.g. a print speed) into another QA Device, typically a printer QA Device. It also allows a system to change that parameter value at some later date.

A parameter upgrader QA Device is able to perform a fixed number of upgrades, and this number is effectively a consumable value. Thus the number of available upgrades decreases by 1 with each upgrade, and can be replenished by a value upgrader QA Device.

4.2.8 Key programmer QA Device

Secret batch keys are inserted into QA Devices during instantiation (e.g. manufacture). These keys must be replaced by the final secret keys when the purpose of the QA Device is known. The *Key Programmer QA Device* implements all necessary functions for replacing keys in other QA Devices.

4.3 Signature

Digital signatures are used throughout the authentication protocols of the QA Chip Logical Interface. A signature is produced by passing data plus a secret key through a keyed hash function. The signature proves that the data was signed by someone who knew the secret key.
The signature function used throughout the QA Chip Logical Interface is HMAC-SHA1 [1].

4.3.4 Authenticated Read

This is a read of data from a non-trusted QA Device that also includes a check of the signature (see Section 4.3.3). When the System determines that the signature is correct for the returned data (e.g. by asking a trusted QA Device to test the signature) then the System is able to trust that the data

has not been tampered en route from the read, and was actually stored on the non-trusted QA Device.

4.3.5 Authenticated Write

5 An authenticated write is a write to the data storage area in a QA Device where the write request includes both the new data and a signature. The signature is based on a key that has write access permissions to the region of data in the QA Device, and proves to the receiving QA Device that the writer has the authority to perform the write. For example, a Value Upgrader Refilling Device is able to authorize a system to perform an authenticated write to upgrade a Consumable QA Device (e.g.

10 to increase the amount of ink in an Ink QA Device).

The QA Device that receives the write request checks that the signature matches the data (so that it hasn't been tampered with en route) and also that the signature is based on the correct authorization key.

An authenticated write can be followed by an authenticated read to ensure (from the system's point

15 of view) that the write was successful.

4.3.6 Non-authenticated Write

A non-authenticated write is a write to the data storage area in a QA Device where the write request includes only the new data (and no signature). This kind of write is used when the system wants to update areas of the QA Device that have no access-protection.

20 The QA Device verifies that the destination of the write request has access permissions that permit anyone to write to it. If access is permitted, the QA Device simply performs the write as requested.

A non-authenticated write can be followed by an authenticated read to ensure (from the system's point of view) that the write was successful.

4.3.7 Authorized Modification of Data

25 Authorized modification of data refers to modification of data via authenticated writes (see Section 4.3.5).

Table 2 provides a summary of the data structures used in the QA Chip Logical Interface..

Table 2. List of data structures

Group description	Name	Represented by	Size	Description
QA Device instance identifier	Chip Identifier	ChipId	48 bits	Unique identifier for this QA Device.
Key and key related data	Number of Keys	NumKeys	8	Number of key slots available in this QA Device.
	Key	K	160 bits per key	K is the secret key used for calculating signatures. K_n is the key stored in the nth key slot.
	Key Identifier	KeyId	31 bits per key	Unique identifier for each key. $KeyId_n$ is the key identifier for the key stored in slot n.
	KeyLock	KeyLock	1bit per key	Flag indicates whether the key is locked in the corresponding slot or not. $KeyLock_n$ is the key lock flag for slot n.
Operating and state data	Number of Memory Vectors	NumVectors	4	Number of 512 bit memory vectors in this QA Device.
	Memory Vector	M	512 bits per M^i	M is a 512 bit memory vector. The 512-bit vector is divided into 16 x 32 bit words.
		M^0		M^0 stores application specific data that is protected by access permissions for key-based and non-key based writes.
		M^1		M^1 stores the attributes for M^0, and is write-once-only.
		M^{2+}		M^{2+} stores application specific data that is protected only by non key-based access permissions.
	Permissions	P_n	16 bits per P	Access permissions for each word of M^{1+}. n = number of M^{1+} vectors
Session data	Random Number	R	160 bits	Current random number used to ensure time varying messages. Changes after each successful authentication or signature generation.

6   Instance/device identifier

Each QA Device requires an identifier that allows unique identification of that QA Device by external systems, ensures that messages are received by the correct QA Device, and ensures that the same device can be used across multiple transactions.

Strictly speaking, the identifier only needs to be unique within the context of a key, since QA Devices only accept messages that are appropriately signed. However it is more convenient to have the instance identifier completely unique, as is the case with this design.

The identifier functionality is provided by *ChipId*.

6.1   CHIPID

ChipId is the unique 64-bit QA Device identifier. The ChipId is set when the QA Device is instantiated, and cannot be changed during the lifetime of the QA Device.
A 64-bit ChipId gives a maximum of 1844674 trillion unique QA Devices.

7   Key and key related data
7.1   NUMKEYS, K, KEYID, AND KEYLOCK
Each QA Device contains a number of secret keys that are used for signature generation and verification. These keys serve two basic functions:

- For reading, where they are used to verify that the read data came from the particular QA Device and was not altered *en route*.
- For writing, where they are used to ensure only authorised modification of data.

Both of these functions are achieved by signature generation; a key is used to generate a signature for subsequent transmission from the device, and to generate a signature to compare against a received signature.

The number of secret keys in a QA Device is given by *NumKeys*. For this version of the QA Chip Logical Interface, NumKeys has a maximum value of 8.

Each key is referred to as K, and the subscripted form K_n refers to the nth key where n has the range 0 to NumKeys-1 (i.e. 0 to 7). For convenience we also refer to the nth key as being the key in the nth *keyslot*.

The length of each key is 160-bits. 160-bits was chosen because the output signature length from the signature generation function (HMAC-SHA1) is 160 bits, and a key longer than 160-bits does not add to the security of the function.

The security of the digital signatures relies upon keys being kept secret. To safeguard the security of each key, keys should be generated in a way that is not deterministic. Ideally each key should be

programmed with a physically generated random number, gathered from a physically random phenomenon. Each key is initially programmed during QA Device instantiation.

Since all keys must be kept secret and must never leave the QA Device, each key has a corresponding 31-bit *KeyId* which can be read to determine the identity or label of the key without

5 revealing the value of the key itself. Since the relationship between keys and KeyIds is 1:1, a system can read all the KeyIds from a QA Device and know which keys are stored in each of the keyslots.

Finally, each keyslot has a corresponding 1-bit *KeyLock* status indicating whether the key in that slot/position is allowed to be replaced (securely replaced, and only if the old key is known). Once a

10 key has been locked into a slot, it cannot be unlocked i.e. it is the final key for that slot. A key can only be used to perform authenticated writes of data when it has been locked into its keyslot (i.e. its KeyLock status = 1). Refer to Section 8.1.1.5 for further details.

Thus each of the NumKeys keyslots contains a 160-bit key, a 31-bit KeyId, and a 1-bit KeyLock.

7.2   COMMON AND VARIANT SIGNATURE GENERATION

15 To create a digital signature, we pass the data to be signed together with a secret key through a key dependent one-way hash function. The key dependent one-way hash function used throughout the QA Chip Logical Interface is HMAC-SHA1[1].

Signatures are only of use if they can be validated i.e. QA Device A produces a signature for data and QA Device B can check if the signature was valid for that particular data. This implies that A

20 and B must share some secret information so that they can generate equivalent signatures.

Common key signature generation is when QA Device A and QA Device B share the exact same key i.e. key K_A = key K_B. Thus the signature for a message produced by A using K_A can be equivalently produced by B using K_B. In other words SIG_{KA}(message) = SIG_{KB}(message) because key K_A = key K_B.

25 *Variant key signature generation* is when QA Device B holds a base key, and QA Device A holds a variant of that key such that K_A = owf(K_B,U_A) where owf is a one-way function based upon the base key (K_B) and a unique number in A (U_A). Thus A can produce SIG_{KA}(message), but for B to produce an equivalent signature it must produce K_A by reading U_A from A and using its base key K_B. K_A is referred to as a *variant key* and K_B is referred to as the *base/common key*. Therefore, B can

30 produce equivalent signatures from many QA Devices, each of which has its own unique variant of K_B. Since ChipId is unique to a given QA Device, we use that as U_A. A one-way function is required to create K_A from K_B or it would be possible to derive K_B if K_A were exposed.

Common key signature generation is used when A and B are equally available[1] to an attacker. For example, Printer QA Devices and Ink QA Devices are equally available to attackers (both are

---

[1]The term "equally available" is relative. It typically means that the ease of availability of both are the effectively the same, regardless of price (e.g. both A and B are commercially available and effectively equally easy to come by).

commonly available to an attacker), so shared keys between these two devices should be common keys.

Variant key signature generation is used when B is not readily available to an attacker, and A is readily available to an attacker. If an attacker is able to determine K_A, they will not know K_A for any other QA Device of class A, and they will not be able to determine K_B.

The QA Device producing or testing a signature needs to know if it must use the common or variant means of signature generation. Likewise, when a key is stored in a QA Device, the status of the key (whether it is a base or variant key) must be stored along with it for future reference. Both of these requirements are met using the KeyId as follows:

The 31-bit KeyId is broken into two parts:

- A 30-bit unique identifier for the key. Bits 30-1 represents the Id.
- A 1-bit Variant Flag, which represents whether the key is a base key or a variant key. Bit 0 represents the Variant Flag.

Table 247 describes the relationship of the Variant Flag with the key.

Table 247. Variant Flag representation

value	Key represented
0	Base key
1	Variant key

7.2.1   Equivalent signature generation between QA Devices

Equivalent signature generation between 4 QA Devices A, B, C and D is shown in Figure 363. Each device has a single key. KeyId.*Id* of all four keys are the same i.e KeyId$_A$.*Id* = KeyId$_B$.*Id* = KeyId$_C$.*Id* = KeyId$_D$.*Id*.

If KeyId$_A$.*VariantFlag* = 0 and KeyId$_B$.*VariantFlag* = 0, then a signature produced by A, can be equivalently produced by B because $K_A = K_B$.

If KeyId$_B$.*VariantFlag* = 0 and KeyId$_C$.*VariantFlag* = 1, then a signature produced by C, is equivalently produced by B because $K_C = f$ (K_B, ChipId$_C$).

If KeyId$_C$.*VariantFlag* = 1 and KeyId$_D$.*VariantFlag* = 1, then a signature produced by C, cannot be equivalently produced by D because there is no common base key between the two devices.

If KeyId$_D$.*VariantFlag* = 1 and KeyId$_A$.*VariantFlag* = 0, then a signature produced by D, can be equivalently produced by A because $K_D = f$ (K_A, ChipId$_D$).

8 Operating and state data

The primary purpose of a QA Device is to securely hold application-specific data. For example if the QA Device is an Ink QA Device it may store ink characteristics and the amount of ink-remaining. If the QA Device is a Printer QA Device it may store the maximum speed and width of printing.

For secure manipulation of data:

- Data must be clearly identified (includes typing of data).

- Data must have clearly defined access criteria and permissions.

The QA Chip Logical Interface contains structures to permit these activities.

The QA Device contains a number of kinds of data with differing access requirements:

- Data that can be decremented by anyone, but only increased in an authorised fashion e.g. the amount of ink-remaining in an ink cartridge.

- Data that can only be decremented in an authorised fashion e.g. the number of times a Parameter Upgrader QA Device has upgraded another QA Device.

- Data that is normally read-only, but can be written to (changed) in an authorised fashion e.g. the operating parameters of a printer.

- Data that is always read-only and doesn't ever need to be changed e.g. ink attributes or the serial number of an ink cartridge or printer.

- Data that is written by QACo/Silverbrook, and must not be changed by the OEM or end user e.g. a licence number containing the OEM's identification that must match the software in the printer.

- Data that is written by the OEM and must not be changed by the end-user e.g. the machine number that filled the ink cartridge with ink (for problem tracking).

8.1 M

M is the general term for all of the memory (or data) in a QA Device. M is further subscripted to refer to those different parts of M that have different access requirements as follows:

- M_0 contains all of the data that is protected by access permissions for key-based (authenticated) and non-key-based (non-authenticated) writes.

- M_1 contains the type information and access permissions for the M_0 data, and has write-once permissions (each sub-part of M_1 can only be written to once) to avoid the possibility of changing the type or access permissions of something after it has been defined.

- M_2, M_3 etc., referred to as M_{2+}, contains all the data that can be updated by anyone until the permissions for those sub-parts of M_{2+} have changed from read/write to read-only.

While all QA Devices must have at least M_0 and M_1, the exact number of memory vectors (M_ns) available in a particular QA Device is given by *NumVectors*. In this version of the QA Chip Logical Interface there are exactly 4 memory vectors, so NumVectors = 4.

762

Each M_n is 512 bits in length, and is further broken into 16 × 32 bit words. The ith word of M_n is referred to as $M_n[i]$. $M_n[0]$ is the least significant word of M_n, and $M_n[15]$ is the most significant word of M_n.

8.1.1 M_0 and M_1

5   In the general case of data storage, it is up to the external accessor to interpret the bits in any way it wants. Data structures can be arbitrarily arranged as long as the various pieces of software and hardware that interpret those bits do so consistently. However if those bits have value, as in the case of a consumable, it is vital that the value cannot be increased without appropriate authorisation, or one type of value cannot be added to another incompatible kind e.g. dollars should

10   never be added to yen.

Therefore M_0 is divided into a number of *fields*, where each field has a size, a position, a type and a set of permissions. M_0 contains all of the data that requires authenticated write access (one data element per field), and M_1 contains the field information i.e. the size, type and access permissions for the data stored in M_0.

15   Each 32-bit word of M_1 defines a field. Therefore there is a maximum of 16 defined fields. $M_1[0]$ defines field 0, $M_1[1]$ defines field 1 and so on. Each field is defined in terms of:

*   size and position, to permit external accessors determine where a data item is

*   type, to permit external accessors determine what the data represents

*   permissions, to ensure approriate access to the field by external accessors.

20   The 32-bit value $M_1[n]$ defines the conceptual field attributes for field n as follows:

With regards to consistency of interpretation, the type, size and position information stored in the various words of M_1 allows a system to determine the contents of the corresponding fields (in M_0) held in the QA Device. For example, a 3-color ink cartridge may have an Ink QA Device that holds the amount of cyan ink in field 0, the amount of magenta ink in field 1, and the amount of yellow ink

25   in field 2, while another single-color Ink QA Device may hold the amount of yellow ink in field 0, where the size of the fields in the two Ink QA Devices are different.

A field must be defined (in M_1) before it can be written to (in M_0). At QA Device instantiation, the whole of M_0 is 0 and no fields are defined (all of M_1 is 0). The first field (field 0) can only be created by writing an appropriate value to $M_1[0]$. Once field 0 has been defined, the words of M_0

30   corresponding to field 0 can be written to (via the appropriate permissions within the field definition $M_1[0]$).

Once a field has been defined (i.e. $M_1[n]$ has been written to), the size, type and permissions for that field cannot be changed i.e. M_1 is write-once. Otherwise, for example, a field could be defined to be lira and given an initial value, then the type changed to dollars.

35   The size of a field is measured in terms of the number of consecutive 32-bit words it occupies. Since there are only 16 × 32-bit words in M_0, there can only be 16 fields when all 16 fields are

defined to be 1 word sized each. Likewise, the maximum size of a field is 512 bits when only a single field is defined, and it is possible to define two fields of 256-bits each.

Once field 0 has been created, field 1 can be created, and so on. When enough fields have been created to allocate all of M_0, the remaining words in M_1 are available for write-once general data storage purposes.

It must be emphasised that when a field is created the permissions for that field are final and cannot be changed. This also means that any keys referred to by the field permissions must be already locked into their keyslots. Otherwise someone could set up a field's permissions that the key in a particular keyslot has write access to that field without any guarantee that the desired key will be ever stored in that slot (thus allowing potential mis-use of the field's value).

8.1.1.1 Field Size and Position

A field's size and position are defined by means of 4 bits (referred to as *EndPos*) that point to the least significant word of the field, with an implied position of the field's most significant word. The implied position of field 0's most significant word is $M_0[15]$. The positions and sizes of all fields can therefore be calculated by starting from field 0 and working upwards until all the words of M_0 have been accounted for.

The default value of $M_1[0]$ is 0, which means field0.endPos = 0. Since field0.startPos = 15, field 0 is the only field and is 16 words long.

8.1.1.1.1 Example

Suppose for example, we want to allocate 4 fields as follows:

- field 0 : 128 bits (4 × 32-bit words)
- field 1: 32 bits (1 × 32-bit word)
- field 2: 160 bits (5 × 32-bit words)
- field 3: 192 bits (6 × 32-bit words)

Field 0's position and size is defined by $M_1[0]$, and has an assumed start position of 15, which means the most significant word of field 0 must be in $M_0[15]$. Field 0 therefore occupies $M_0[12]$ through to $M_0[15]$, and has an endPos value of 12.

Field 1's position and size is defined by $M_1[1]$, and has an assumed start position of 11 (i.e. $M_1[0]$.endPos - 1). Since it has a length of 1 word, field 1 therefore occupies only $M_0[11]$ and its end position is the same as its start position i.e. its endPos value is 11.

Likewise field 2's position and size is defined by $M_1[2]$, and has an assumed start position of 10 (i.e. $M_1[1]$.endPos - 1). Since it has a length of 5 words, field 2 therefore occupies $M_0[6]$ through to $M_0[10]$ and and has an endPos value of 6.

Finally, field 3's position and size is defined by $M_1[3]$, and has an assumed start position of 5 (i.e. $M_1[2]$.endPos - 1). Since it has a length of 6 words, field 3 therefore occupies $M_0[5]$ through to $M_0[0]$ and and has an endPos value of 0.

Since all 16 words of M_0 are now accounted for in the 4 fields, the remaining words of M_1 (i.e. $M_1[4]$ though to $M_1[15]$) are ignored, and can be used for any write-once (and thence read-only) data.

Figure 365 shows the same example in diagramatic format.

8.1.1.1.2 Determining the number of fields

The following pseudocode illustrates a means of determining the number of fields:

```
fieldNum FindNumFields(M1)

startPos ← 15

fieldNum ← 0

While (fieldNum < 16)

    endPos ← M1[fieldNum].endPos

    If (endPos > startPos)

        # error in this field... so must be an attack

        attackDetected() # most likely clears all keys and data

    EndIf

    fieldNum++

    If (endPos = 0)

        return fieldNum  # is already incremented

    Else

        startPos ← endPos - 1   # endpos must be > 0

    EndIf

EndWhile

# error if get here since 16 fields are consumed in 16 words at
most

attackDetected() # most likely clears all keys and data
```

8.1.1.1.3 Determining the sizes of all fields

The following pseudocode illustrates a means of determing the sizes of all valid fields:

```
FindFieldSizes(M1, fieldSize[])

numFields ← FindNumFields(M1)  # assumes that FindNumFields does
all checking

ntartPos ← 15

fieldNum ← 0

While (fieldNum < numFields)

    EndPos ← M1[fieldNum].endPos

    fieldSize[fieldNum] = startPos - endPos + 1

    startPos ← endPos - 1   # endpos must be > 0

    fieldNum++
```

```
            EndWhile
            While (fieldNum < 16)
                fieldSize[fieldNum]  ← 0
                fieldNum++
5           EndWhile
```

8.1.1.2 Field Type

The system must be able to identify the type of data stored in a field so that it can perform
operations using the correct data. For example, a printer system must be able identify which of a
consumable's fields are ink fields (and which field is which ink) so that the ink usage can be
10 correctly applied during printing.

A field's type is defined by 15 bits. Table 332 in Appendix A lists the field types that are specifically
required by the QA Chip Logical Interface and therefore apply across all applications.

The default value of $M_1[0]$ is 0, which means field0.type = 0 (i.e. non-initialised).

Strictly speaking, the type need only be interpreted by all who can securely read and write to that
15 field i.e. within the context of one or more keys. However it is convenient if possible to keep all types
unique for simplistic identification of data across all applications.

In the general case, an external system communicating with a QA Device can identify the data
stored in M0 in the following way:

- Read the *KeyId* of the key that has permission to write to the field. This will a give broad
20   identification of the data type, which may be sufficient for certain applications.

- Read the type attribute for the field to narrow down the identity within the broader context of
    the KeyId.

For example, the printer system can read the *KeyId* to deduce that the data stored in a field can be
written to via the HP_Network_InkRefill key, which means that any data is of the general ink
25 category known to HP Network printers. By further reading the type attribute for the field the system
can determine that the ink is Black ink.

8.1.1.3 Field Permissions

All fields can be ready by everyone. However writes to fields are governed by 13-bits of permissions
that are present in each field's attribute definition. The permissions describe who can do what to a
30 specific field.

Writes to fields can either be authenticated (i.e. the data to be written is signed by a key and this
signature must be checked by the receiving device before write access is given) or non-
authenticated (i.e. the data is not signed by a key). Therefore we define a single bit (*AuthRW*) that
specifies whether authenticated writes are permitted, and a single bit (*NonAuthRW*) specifying
35 whether non-authenticated writes are permitted. Since it is pointless to permit both authenticated
and non-authenticated writes to write any value (the authentciated writes are pointless), we further
define the case when both bits are set to be interpreted as authenticated writes are permitted, but

non-authenticated writes only succeed when the new value is less than the previous value i.e. the permission is *decrement-only*. The interpretation of these two bits is shown in Table 249.

Table 249. Interpretation of AuthRW and NonAuthRW

NonAuthRW	AuthRW	Interpretation
0	0	Read-only access (no-one can write to this field). This is the initial state for each field. At instantiation all of M_1 is 0 which means AuthRW and NonAuthRW are 0 for each field, and hence none of M_0 can be written to until a field is defined.
0	1	Authenticated write access is permitted Non-authenticated write acecss is not permitted
1	0	Authenticated write access is not permitted Non-authenticated write access is permitted (i.e. anyone can write to this field)
1	1	Authenticated write access is permitted Non-authenticated write access is decrement-only.

If authenticated write access is permitted, there are 11 additional bits (bringing the total number of permission bits to 13) to more fully describe the kind of write access for each key. We only permit a single key to have the ability to write any value to the field, and the remaining keys are defined as being either not permitted to write, or as having decrement-only write access. A 3-bit *KeyNum* represents the slot number of the key that has the ability to write any value to the field (as long as the key is locked into its key slot), and an 8-bit *KeyPerms* defines the write permissions for the (maximum of) 8 keys as follows:

- *KeyPerms[n] = 0*: The key in slot n (i.e. K_n) has no write access to this field (except when n = KeyNum). Setting KeyPerms to 0 prohibits a key from transferring value (when an amount is deducted from field in one QA Device and transferred to another field in a different QA Device)

- *KeyPerms[n] = 1*: The key in slot n (i.e. K_n) is permitted to perform decrement-only writes to this field (as long as K_n is locked in its key slot). Setting KeyPerms to 1 allows a key to transfer value (when an amount is deducted from field in one QA Device and transferred to another field in a different QA Device).

The 13-bits of permissions (within bits 4-16 of $M_1[n]$) are allocated as follows:

8.1.1.3.1 Example 1

Figure 367 shows an example of permission bits for a field.

In this example we can see:

- *NonAuthRW* = 0 and *AuthRW* = 1, which means that only authenticated writes are allowed i.e. writes to the field without an appropriate signature are not permitted.

- *KeyNum* = 3, so the only key permitted to write any value to the field is key 3 (i.e. K_3).

- *KeyPerms[3]* = 0, which means that although key 3 is permitted to write to this field, key 3 can't be used to transfer value from this field to other QA Devices.

- *KeyPerms[0,4,5,6,7]* = 0, which means that these respective keys cannot write to this field.

- *KeyPerms[1,2]* = 1, which means that keys 1 and 2 have decrement-only access to this field i.e. they are permitted to write a new value to the field only when the new value is less than the current value.

8.1.1.3.2 Example 2

Figure 368 shows a second example of permission bits for a field.

In this example we can see:

- *NonAuthRW* and *AuthRW* = 1, which means that authenticated writes are allowed and writes to the field without a signature are only permitted when the new value is less than the current value (i.e. non-authenticated writes have decrement-only permission).

- *KeyNum* = 3, so the only key permitted to write any value to the field is key 3 (i.e. K_3).

- *KeyPerms[3]* = 1, which means that key 3 is permitted to write to this field, and can be used to transfer value from this field to other QA Devices.

- *KeyPerms[0,4,5,6,7]* = 0, which means that these respective keys cannot write to this field.

- *KeyPerms[1,2]* = 1, which means that keys 1 and 2 have decrement-only access to this field i.e. they are permitted to write a new value to the field only when the new value is less than the current value.

8.1.1.4 Summary of Field attributes

Figure 369 shows the breakdown of bits within the 32-bit field attribute value $M_1[n]$. Table 250 summarises each attribute.

Table 250. Attributes for a field

Attribute	Sub-attribute name	Size in bits	Interpretation
Type	Type	15	Gives additional identification of the data

			stored in the field within the context of the accessors of that field.
Permissions	KeyNum	3	The slot number of the key that has authenticated write access to the field.
	NonAuthRW	1	0 = non-authenticated writes are not permitted to this field. 1 = non-authenticated writes are permitted to this field (see Table 249).
	AuthRW	1	0 = authenticated writes are not permitted to this field. 1 = authenticated writes are permitted to this field.
	KeyPerms	8	Bitmap representing the write permissions for each of the keys when AuthRW = 1. For each bit: 0 = no write access for this key (except for key KeyNum) 1 = decrement-only access is permitted for this key.
Size and Position	EndPos	4	The word number in M_0 that holds the lsw of the field. The msw is held in M1[fieldNum-1], where msw of field 0 is 15.

8.1.1.5 Permissions of M_1

M_1 holds the field attributes for data stored in M_0, and each word of M_1 can be written to once only. It is important that a system can determine which words are available for writing. While this can be determined by reading M_1 and determining which of the words is non-zero, a 16-bit permissions value P_1 is available, with each bit indicating whether or not a given word in M_1 has been written to. Bit n of P_1 represents the permissions for $M_1[n]$ as follows:

Table 251. Interpretation of $P_1[n]$ i.e. bit n of M_1's permission

	Description
0	writes to $M_1[n]$ are not permitted i.e. this word is now read-only
1	writes to $M_1[n]$ are permitted

Since M_1 is write-once, whenever a word is written to in M_1, the corresponding bit of P_1 is also cleared, i.e. writing to $M_1[n]$ clears $P_1[n]$.

Writes to $M_1[n]$ only succeed when all of $M_1[0...n-1]$ have already written to (i.e. previous fields are defined) i.e.

5 • $M_1[0..n-1]$ must have already been written to (i.e. $P_1[0..n-1]$ are 0)

• $P_1[n] = 1$ (i.e. it has not yet been written to)

In addition, if $M_1[n-1].endPos \neq 0$, the new $M_1[n]$ word will define the attributes of field n, so must be further checked as follows:

•   The new $M_1[n].endPos$ must be valid (i.e. must be less than $M_1[n-1].endPos$)

10 •   If the new $M_1[n].authRW$ is set, K_{keyNum} must be locked, and all keys referred to by the new $M_1[n].keyPerms$ must also be locked.

However if $M_1[n-1].endPos = 0$, then all of M_0 has been defined in terms of fields. Since enough fields have been created to allocate all of M_0, any remaining words in M_1 are available for write-once general data storage purposes, and are not checked any further.

15   8.1.2   M2+

M_2, M_3 etc., referred to as M_{2+}, contains all the data that can be updated by anyone (i.e. no authenticated write is required) until the permissions for those sub-parts of M_{2+} have changed from read/write to read-only.

The same permissions representation as used for M_1 is also used for M_{2+}. Consequently P_n is a 16-

20 bit value that contains the permissions for M_n (where n > 0). The permissions for word w of M_n is given by a single bit $P_n[w]$. However, unlike writes to M_1, writes to M_{2+} do not automatically clear bits in P. Only when the bits in P_{2+} are explictly cleared (by anyone) do those corresponding words become read-only and final.

9   Session data

25 Data that is valid only for the duration of a particular communication session is referred to as session data. Session data ensures that every signature contains different data (sometimes referred to as a *nonce*) and this prevents replay attacks.

9.1   R

R is a 160-bit random number seed that is set up (when the QA Device is instantiated) and from

30 that point on it is internally managed and updated by the QA Device. R is used to ensure that each signed item contains time varying information (not chosen by an attacker), and each QA Device's R is unrelated from one QA Device to the next.

This R is used in the generation and testing of signatures.

An attacker must not be able to deduce the values of R in present and future devices. Therefore, R

35 should be programmed with a cryptographically strong random number, gathered from a physically random phenomenon (must not be deterministic).

9.2   ADVANCING R

The session component of the message must only last for a single session (challenge and response).

The rules for updating R are as follows:

5   • Reads of R do not advance R.

  • Everytime a signature is produced with R, R is advanced to a new random number.

  • Everytime a signature including R is tested and is found to be correct, R is advanced to a new random number.

9.3 R_L AND R_E

10   Each signature contains 2 pieces of session data i.e. 2 Rs:

  • One R comes from the QA Device issuing the challenge i.e. the challenger. This is so the challenger can ensure that the challenged QA Device isn't simply replaying an old signature i.e. the challenger is protecting itself against the challenged.

  • One R comes from the device responding to the challenge i.e. the challenged. This is so the
15   challenged never signs anything that is given to it without inserting some time varying change i.e. protects the challenged from the challenger in case the challenger is actually an attacker performing a chosen text attack

Since there are two Rs, we need to distinguish between them. We do so by defining each R as external (R_E) or local (R_L) depending on its use in a given function. For example, the challenger
20   sends out its local R, referred to as R_L. The device being challenged receives the challenger's R as an external R, i.e R_E. It then generates a signature using its R_L and the challenger's R_E. The resultant signature and R_L are sent to the challenger as the response. The challenger receives the signature and R_E (signature and R_L produced by the device being challenged), produces its own signature using R_L (sent to the device being challenged earlier) and R_E received, and compares that
25   signature to the signature received as response.

SIGNATURE FUNCTIONS

10   Objects

10.1   KEYREF

10.1.1   Object description

30   Instead of passing keys directly into a function, a *KeyRef* (i.e. key reference) object is passed instead. A KeyRef object encapsulates the process by which a key is formed for common and variant forms of signature generation (based on the setting of the variables within the object). A KeyRef defines which key to use, whether it is a common or variant form of that key, and, if it is a variant form, the ChipId to use to create the variant. For more information about common and
35   variant forms of keys, see Section 7.2.

Users pass KeyRef objects in as input parameters to public functions of the QA Chip Logical Interface , and these KeyRefs are subsequently passed to the signature function (called within the

interface function). Note, however, that the method functions for KeyRef objects are not available outside the QA Chip Logical Interface.

10.1.2 Object variables

Table 252 describes each of the variables within a KeyRef object.

Table 252. Description of object variables for KeyRef object

Parameter	Description
keyNum	Slot number of the key to use as the basis for key formation
useChipId	0 = the key to be formed is a common key (i.e. is the same as K_{keyNum}) 1 = the key to be formed is a variant key based on K_{keyNum}
ChipId	When useChipId = 1, this is the ChipId to be used to form the variant key (this will be the ChipId of the QA Device which stores the variant of K_{keyNum}) When useChipId = 0, chipId is not used

10.1.3 Object Methods

10.1.3.1 getKey

public key getKey(void)

10.1.3.1.1 Method description

This method is a public method (public in object oriented terms, not public to users of the QA Chip Logical Interface) and is called by the *GenerateSignature* function to return the key for use in signature generation.

If *useChipId* is true, the *formKeyVariant* method is called to form the key using *chipId* and then return the variant key. If *useChipId* is false, the key stored in slot *keyNum* is returned.

10.1.3.1.2 Method sequence

The *getKey* method is illustrated by the following pseudocode:

```
If (useChipId = 0)
    key ← K_keyNum
Else
    key ← formKeyVariant()
EndIf
Return key
```

10.1.3.2 formKeyVariant

private key formKeyVariant (void)

10.1.3.2.1 Method description

This method produces the variant form of a key, based on the K_{keyNum} and *chipId*. As described in Section 7.2, the variant form of key K_{keyNum} is generated by owf (K_{keyNum}, chipId) where owf is a one-way function.

In addition, the time taken by owf must not depend on the value of the key i.e. the timing should be effectively constant. This prevents timing attacks on the key.

At present, owf is SHA1, although this still needs to be verified. Thus the variant key is defined to be SHA1(K_{keyNum} | chipId).

5   10.1.3.2.2   Method sequence

The *formKeyVariant* method is illustrated by the following pseudocode:

```
key ← SHA1( KkeyNum | chipId) # Calculation must take constant time
Return key
```

11   Functions

10   Digital signatures form the basis of all authentication protocols within the QA Chip Logical Interface . The signature functions are not directly available to users of the QA Chip Logical Interface , since a golden rule of digital signatures is never to sign anything exactly as it has been given to you. Instead, these signature functions are internally available to the functions that comprise the public interface, and are used by those functions for the formation of keys and the generation of

15   signatures.

11.1   GENERATESIGNATURE

Input:	*KeyRef, Data, Random1, Random2*
Output:	*SIG*
Changes:	*None*
Availability:	*All devices*

11.1.1   Function description

This function uses *KeyRef* to obtain the actual key required for signature generation, appends *Random1* and *Random2* to *Data*, and performs HMAC_SHA1[key, Data] to output a signature. HMAC_SHA1 is described in [1]. In addition, this operation must take constant time irrespective of

25   the value of the key (see Section 10.1.3.2 for more details).

11.1.2   Input parameter description

Table 253 describes each of the input parameters:

Table 253. Description of input parameters for GenerateSignature

Parameter	Description
KeyRef	This is an instance of the KeyRef object for use by the GenerateSignature function. *For common key signature generation: KeyRef.keyNum* = Slot number of the key to be used to produce the signature. *KeyRef.useChipId* = 0
	For variant key signature generation: KeyRef.keyNum = Slot number of the key to be used for generating the variant key, where the var iant key is to be used to produce the signature *KeyRef.useChipId* = 1 *KeyRef.chipId* = ChipId of the QA Device which stores the variant of $K_{KeyRef.keyNum}$, and uses the variant key for

	signature generation.
Data	Preformatted data to be signed. Random1 and Random2 are appended to Data before the signature is generated to ensure that the signature is session based (applicable only to a single session).
Random1	This is the session component from the QA Device that is responding to the challenge.
Random2	This is the session component from the QA Device that issued the challenge.

11.1.3 Output parameter description

Table 254 describes each of the output parameters.

Table 254. Description of output parameters for GenerateSignature

Parameter	Description
SIG	SIG = SIG_{key}(Data | Random1 | Random2) where key = KeyRef.getKey()

11.1.4 Function sequence

The *GenerateSignature* function is illustrated by the following pseudocode:

```
key ← KeyRef.getKey()

dataToBeSigned ← Data|Random1|Random2

SIG ← HMAC_SHA1(key, dataToBeSigned) # Calculation must take
constant time

Output SIG

Return
```

BASIC FUNCTIONS

12 Definitions

This section defines return codes and constants referred to by functions and pseudocode.

12.1 RESULTFLAG

The *ResultFlag* is a byte that indicates the return status from a function. Callers can use the value of ResultFlag to determine whether a call to a function succeeded or failed, and if the call failed, the specific error condition.

Table 255 describes the *ResultFlag* values and the mnemonics used in the pseudocode.

Table 255. ResultFlag value description

Mnemonic	Description	Possible causes
Pass	Function completed sucessfully	Function successfully completed requested task.
Fail	General Failure	An error occurred during function processing.
BadSig	Signature mismatch	Input signature didn't match the generated signature.
InvalidKey	KeyRef incorrect	Input *KeyRef.keyNum* > 3.
InvalidVector	VectNum incorrect	Input $M_{VectNum}$> 3.
InvalidPermission	Permission not adqeuate to per form operation.	Trying to perform a *Write* or *WriteAuth* with incorrect permissions.
KeyAlreadyLocked	Key already locked .	Key cannot be changed because it has already been locked.

12.2 CONSTANTS

5   Table 256 describes the constants referred to by functions and pseudocode.

Table 256. Constants

Definition	Value
MaxKey	NumKeys -1 (typically 7)
MaxM	NumVectors -1 (typically 3)
MaxWordIn M	16 - 1 = 15

13   GetInfo

Input: *None*

10   *Output:*   *ResultFlag, SoftwareReleaseIdMajor, SoftwareReleaseIdMinor,*
NumVec tors,   NumKeys,ChipId
DepthOfRollBackCache (for an upgrade device only)

Changes   :   None

Availability: All devices

15   ## 13.1   FUNCTION DESCRIPTION

Users of QA Devices must call the *GetInfo* function on each QA Device before calling any other functions on that device.

The *GetInfo* function tells the caller what kind of QA Device this is, what functions are available and what properties this QA Device has. The caller can use this information to correctly call functions with appropriately formatted parameters.

The first value returned, *SoftwareReleaseIdMajor*, effectively identifies what kind of QA Device this is, and therefore what functions are available to callers. *SoftwareReleaseIdMinor* tells the caller which version of the specific type of QA Device this is. The mapping between the *SoftwareReleaseIdMajor* and type of device and their different functions is described in Table 258 Every QA Device also returns *NumVectors*, *NumKeys* and *ChipId* which are required to set input parameter values for commands to the device.

Additional information may be returned depending on the type of QA Device. The *VarDataLen* and *VarData* fields of the output hold this additional information.

13.2 OUTPUT PARAMETERS

Table 257 describes each of the output parameters.

Table 257. Description of output parameters for GetInfo function

Parameter	#bytes	Description
ResultFlag		Indicates whether the function completed successfully or not. If it did not complete successfully, the reason for the failure is returned here. See Section 12.1.
SoftwareReleaseIdMajor	1	This defines the function set that is available on this QA Device.
SoftwareReleaseIdMinor	1	This defines minor software releases within a major release, and are incremental changes to the software mainly to deal with bug fixes.
NumVectors	1	Total number of memory vectors in this QA Device.
NumKeys	1	Total number of keys in this QA Device.
ChipId	6	This QA Device's ChipId
VarDataLen	1	Length of bytes to follow.
VarData	(VarDataLen bytes)	This is additional application specific data, and will be of length *VarDataLen* (i.e. may be 0).

Table 258 shows the mapping between the *SoftwareReleaseIdMajor*, the type of QA Device and the available device functions.

Table 258. Mapping between SoftwareReleaseIdMajor and available device functions

SoftwareReleased Major	Device description	Functions available
1	Ink or Printer QA Device	GetInfo
		Random
		Read
		Test
		Translate
		WriteM1+
		WriteFields
		WriteFieldsAuth
		SetPerm
		ReplaceKey
2	Value Upgrader QA Device (e.g. Ink Refill QA Device)	All functions in the *Ink or Printer Device*, plus:
		StartXfer
		XferAmount
		StartRollBack
		RollBackAmount
3	Parameter Upgrader QA Device	All functions in the *Ink or Printer device*, plus:
		StartXfer
		XferField
		StartRollBack
		RollBackField
4	Key Replacement device	All functions in the *Ink or Printer Device*, plus:
		GetProgramKey
		ReplaceKey - is different from the *Ink or Printer device*
5	Trusted device	All functions in the *Ink or Printer Device*, plus:
		SignM

Table 259 shows the *VarData* components for Value Upgrader and Parameter Upgrader QA Devices.

Table 259. VarData for Value and Parameter Upgrader QA Devices

VarData Components	Length in bytes	Description
DepthOfRollBackCache	*1*	The number of datasets that can be accommodated in the Xfer Entry cache of the device.

13.3 FUNCTION SEQUENCE

The *GetInfo* command is illustrated by the following pseudocode:

```
Output SoftwareReleaseIdMajor

Output SoftwareReleaseIdMinor

Output NumVectors

Output NumKeys

Output ChipId

VarDataLen ← 1 # In case of an upgrade device

Output DepthOfRollBackCache

Return
```

14 Random

Input:	*None*
Output:	R_L
Changes:	*None*
Availability:	*All devices*

The *Random* command is used by the caller to obtain a session component (challenge) for use in subsequent signature generation.

If a caller calls the *Random* function multiple times, the same output will be returned each time. R_L (i.e. this QA Device's R) will only advance to the next random number in the sequence after a successful test of a signature or after producing a new signature. The same R_L can never be used to produce two signatures from the same QA Device.

The Random command is illustrated by the following pseudocode:

```
Output R_L

Return
```

15 Read

Input:	*KeyRef, SigOnly, MSelect, KeyIdSelect, WordSelect, R_E*
Output:	*ResultFlag, SelectedWordsOfSelectedMs, SelectedKeyIds, R_L, SIG_{out}*

Changes: R_L

Availability: All devices

15.1 FUNCTION DESCRIPTION

The *Read* command is used to read data and keyIds from a QA Device. The caller can specify which words from M and which KeyIds are read.

The *Read* command can return both data and signature, or just the signature of the requested data. Since the return of data is based on the caller's input request, it prevents unnecessary information from being sent back to the caller. Callers typically request only the signature in order to confirm that locally cached values match the values on the QA Device .

The data read from an untrusted QA Device (A) using a *Read* command is validated by a trusted QA Device (B) using the *Test* command. The R_L and SIG_{out} produced as output from the *Read* command are input (along with correctly formatted data) to the *Test* command on a trusted QA Device for validation of the signature and hence the data. SIG_{out} can also optionally be passed through the *Translate* command on a number of QA Devices between *Read* and *Test* if the QA Devices A and B do not share keys.

15.2 INPUT PARAMETERS

Table 260 describes each of the input parameters:

Table 260. Description of input parameters for Read

Parameter	Description
KeyRef	*For common key signature generation: KeyRef.keyNum = Slot number of the key to be used for producing the output signature. KeyRef.useChipId = 0*
	No variant key signature generation required
SigOnly	Flag indicating return of signature and data. 0- indicates both the signature and data are to be returned. 1- indicates only the signature is to be returned.
Mselect	Selection of memory vectors to be read - each bit corresponding to a given memory vec tor (a maximum of NumVector bits) 0- indicates the memory vector must not be read. 1- indicates memory vector must be read.
KeyIdSelect	Selection of KeyIds to be read - each bit corresponds to a given KeyId (a maximum of NumKey bits). 0- indicates KeyId must not be read. 1- indicates KeyId must be read.

WordSelect	Selection of words read from a desired M as requested in MSelect. Each WordSelect is 16 bits corresponding to each bit in MSelect. Each bit in the WordSelect indicates whether or not to read the corresponding word for the particular M. 0- indicates word must not be read. 1- indicates word must be read.
R_E	External random value required for output signature generation (i.e the challenge). R_E is obtained by calling the Random function on the device which will receive the SIG_{out} from the Read function.

15.3 OUTPUT PARAMETERS

Table 261 describes each of the output parameters.

5

Parameter	Description
ResultFlag	Indicates whether the function completed successfully or not. If it did not complete successfully, the reason for the failure is returned here. See Section 12.1.
SelectedWordsOfSelectedMs	Selected words from selected memory vectors as requested by MSelect and WordSelect.
SelectedKeyIds	Selected KeyIds as requested by KeyIdSelect.
R_L	Local random value added to the output signature(i.e SIG_{out}).Refer to Figure 370.
SIG_{out}	$SIG_{out} = SIG_{KeyRef}(data \mid R_L \mid R_E)$ as shown in Figure 8. Refer to Section 10.1.3.1 for details.

15.3.1 SIG$_{out}$

Figure 370 shows the formatting of data for output signature generation.

Table 262 gives the parameters included in SIG_{out}

10

Parameter	Length in bits	Value set internally	Value set from Input
RWSense	3	read constant = 000 Refer to Section 15.3.1.1	
MSelect	4		●
KeyIdSelect	8		●

780

ChipId	48	This QA Device's ChipId	
WordSelect	16 per M		●
SelectedWordsOfSelectedMs	32 per word	The appropriate words from the various Ms as selected by the caller	●
R_L	160	This QA Device's current R	
R_E	160		●

15.3.1.1 RWSense

An *RWSense* value is present in the signed data to distinguish whether a signature was produced from a *Read* or produced for a *WriteAuth*.

The *RWSense* is set to a read constant (000) for producing a signature from a read function. The *RWSense* is set to a write constant (001) for producing a signature for a write function.

The *RWSense* prevents signatures produced by *Read* to be subsequently sent into a *WriteAuth* function. Only signatures produced with *RWSense* set to write (001), are accepted by a write function.

15.4 FUNCTION SEQUENCE

The *Read* command is illustrated by the following pseudocode:

```
Accept input parameters- KeyRef, SigOnly, MSelect, KeyIdSelect
# Accept input parameter WordSelect based on MSelect


For i ← 0 to MaxM
    If(MSelect[i] = 1)
        Accept next WordSelect
        WordSelectTemp[i] ← WordSelect
    EndIf
EndFor
Accept R_E


Check range of KeyRef.keyNum
If invalid
    ResultFlag ← InvalidKey
    Output ResultFlag
    Return
EndIf


#Build SelectedWordsOfSelectedMs
```

```
k ← 0 # k stores the word count for SelectedWordsOfSelectedMs
SelectedWordsOfSelectedMs[k] ← 0
For i ← 0 to 3
   If(MSelect[i] = 1)
      For j ← 0 to MaxWordInM
         If(WordSelectTemp[i][j] = 1)
            SelectedWordsOfSelectedMs[k] ← (Mᵢ[j])
            k++
         EndIf
      EndFor
   EndIf
EndFor


#Build SelectedKeyIds
l ← 0 # l stores the word count for SelectedKeyIds
SelectedKeyIds[l] ← 0
For i ← 0 to MaxKey
   If(KeyIdSelect[i] = 1)
      SelectedKeyIds[l] ← KeyId[i]
      l++
   EndIf
EndFor




#Generate message for passing into the GenerateSignature function
data ← (RWSense|MSelect|KeyIdSelect|ChipId|WordSelect
          |SelectedWordsOfSelectedMs|SelectedKeyIds) # Refer to
Figure 370.
#Generate Signature function
SIG_L ← GenerateSignature(KeyRef,data,R_L,R_E) # See Section 11.1
Update R_L to R_L2
ResultFlag ← Pass
Output ResultFlag
If(SigOnly = 0)
   Output SelectedWordsOfSelectedMs, SelectedKeyIds
EndIf
```

```
Output R_L, SIG_L
Return
```

16   Test

Input:	*KeyRef, DataLength, Data, R_E, SIG_E*
Output:	*ResultFlag*
Changes:	R_L
Availability:	*All devices except ink device*

16.1   FUNCTION DESCRIPTION

The *Test* command is used to validate data that has been read from an untrusted QA Device according to a digital signature SIG_E. The data will typically be memory vector and KeyId data. SIG_E (and its related R_E) is the most recent signature - this will be the signature produced by *Read* if *Translate* was not used, or will be the output from the most recent *Translate* if *Translate* was used. The *Test* function produces a local signature (SIG_L= $SIG_{key}(Data|R_E|R_L)$) and compares it to the input signature (SIG_E). If the two signatures match the function returns 'Pass', and the caller knows that the data read can be trusted.

The key used to produce SIG_L depends on whether SIG_E was produced by a QA Device sharing a common key or a variant key. The *KeyRef* object passed into the interface must be set appropriately to reflect this.

The *Test* function accepts preformatted data (as *DataLength* number of words), and appends the external R_E and local R_L to the preformatted data to generate the signature as shown in Figure 371.

16.2   INPUT PARAMETERS

Table 263 describes each of the input parameters.

Table 263. Description of input parameters for Test

Parameter	Description
KeyRef	*For testing common key signature: KeyRef.keyNum* = Slot number of the key to be used for testing the signature. SIG_E produced using $K_{KeyRef.keyNum}$ by the external device. *KeyRef.useChipId* = 0
	For testing variant key signature: KeyRef.keyNum = Slot number of the key to be used for generating the variant key. SIG_E produced using a variant of $K_{KeyRef.keyNum}$ by the external device. *KeyRef.useChipId* = 1 *KeyRef.chipId* = ChipId of the device which generated SIG_E using a variant of $K_{KeyRef.keyNum}$.
DataLength	Length of preformatted data in words. Must be non zero.
Data	Preformatted data to be used for producing the signature.
R_E	External random value required for verifying the input signature. This will be the *R* from the input signature generator (i.e the device generating SIG_E).
SIG_E	External signature required for authenticating input data as shown in Figure 371.

| | The external signature is generated either by a *Read* function or a *Translate* function. A correct $SIG_E = SIG_{KeyRef}(Data \mid R_E \mid R_L)$. |

16.2.1 Input signature verification data format

Figure 371 shows the formatting of data for input signature verification.

The data in Figure 371 (i.e. not R_E or R_L) is typically output from a *Read* function (formatted as per Figure 370). The data may also be generated in the same format by the system from its cache as will be the case when it performs a *Read* using *SigOnly* = 1.

16.3 OUTPUT PARAMETERS

Table 264 describes each of the output parameters.

Table 264. Description of output parameters for Test

Parameter	Description
ResultFlag	Indicates whether the function completed successfully or not. If it did not complete successfully, the reason for the failure is returned here. See Section 12.1.

16.4 FUNCTION SEQUENCE

The *Test* command is illustrated by the following pseudocode:

Accept input parameters- KeyRef, DataLength

```
# Accept input parameter- Data based on DataLength


For i ← 0 to (DataLength - 1)
    Accept next word of Data
EndFor


Accept input parameters - Rᴇ, SIGᴇ


Check range of KeyRef.keyNum
If invalid
    ResultFlag ← InvalidKey
    Output ResultFlag
    Return
EndIf
```

```
                #Generate signature
                SIGₗ ← GenerateSignature(KeyRef,Data,Rₑ,Rₗ)  # Refer to Figure 371.


                #Check signature
5               If(SIGₗ = SIGₑ)
                    Update Rₗ to Rₗ₂
                    ResultFlag ← Pass
                Else
                    ResultFlag ← BadSig
10              EndIf
                Output ResultFlag
                Return
```

17   Translate

Input:	*InputKeyRef, DataLength, Data, R_E, SIG_E, OutputKeyRef, R_{E2}*
Output:	*ResultFlag, R_{L2}, SIG_{Out}*
Changes:	*R_L*
Availability:	*Printer device, and possibly on other devices*

17.1   FUNCTION DESCRIPTION

It is possible for a system to call the *Read* function on QA Device A to obtain data and signature, and then call the *Test* function on QA Device B to validate the data and signature. In the same way it is possible for a system to call the *SignM* function on a trusted QA Device B and then call the *WriteAuth* function on QA Device B to actually store data on B. Both of these actions are only possible when QA Devices A and B share secret key information.

If however, A and B do not share secret keys, we can create a validation chain (and hence extension of trust) by means of translation of signatures. A given QA Device can only translate signatures if it knows the key of the previous stage in the chain as well as the key of the next stage in the chain. The *Translate* function provides this functionality.

The *Translate* function translates a signature from one based on one key to one based another key. The *Translate* function first performs a test of the input signature using the *InputKeyRef,* and if the test succeeds produces an output signature using the *OutputKeyRef.* The *Translate* function can therefore in some ways be considered to be a combination of the *Test* and *Read* function, except that the data is input into the QA Device instead of being read from it.

The *InputKeyRef* object passed into *Translate* must be set appropriately to reflect whether *SIG_E* was produced by a QA Device sharing a common key or a variant key.

The key used to produce output signature *SIG_{out}* depends on whether the translating device shares a common key or a variant key with the QA Device receiving the signature. The *OutputKeyRef* object passed into *Translate* must be set appropriately to reflect this.

Since the *Translate* function does not interpret or generate the data in any way, only preformatted data can be passed in. The *Translate* function does however append the external R_E and local R_L to the preformatted data for verifying the input signature, then advances R_L to R_{L2}, and appends R_{L2} and R_{E2} to the preformatted data to produce the output signature. This is done to protect the keys and prevent replay attacks.

The *Translate* functions translates:

- signatures for subsequent use in *Test*, typically originating from *Read*
- signatures for subsequent use in *WriteAuth*, typically originating from *SignM*

In both cases, preformatted data is passed into the *Translate* function by the system. For translation of data destined for *Test*, the data should be preformatted as per Figure 370 (all words except the Rs). For translation of signatures for use in *WriteAuth*, the data should be preformatted as per Figure 373 (all words except the Rs).

17.2 INPUT PARAMETERS

Table 265 describes each of the input parameters.

Table 265. Description of input parameters for Translate

Parameter	Description
InputKeyRef	*For translating common key input signature: InputKeyRef.keyNum* = Slot number of the key to be used for testing the signature. SIG_E produced using $K_{InputKeyRef.keyNum}$ by the external device. *InputKeyRef.useChipId* = 0
	For translating variant key input signatures: InputKeyRef.keyNum = Slot number of the key to be used for generating the variant key. SIG_E produced using a variant of $K_{InputKeyRef.keyNum}$ by the external device. *InputKeyRef.useChipId* = 1 *InputKeyRef.chipId* = ChipId of the device which generated SIG_E using a variant of $K_{InputKeyRef.keyNum}$.
DataLength:	Length of data in words.
Data	Data used for testing the input signature and for producing the output signature.
R_E	External random value required for verifying input signature. This will be the R from the input signature generator (i.e device generating SIG_E).
SIG_E	External signature required for authenticating input data. The external signature is either generated by a *Read* function, a *Xfer/Rollback* function or a *Translate* function. A correct SIG_E = SIG_{KeyRef}(Data | R_E | R_L).
OutputKeyRef	*For generating common key output signature: OutputKeyRef.keyNum* = Slot number of the key for producing the output signature. *SIGout* produced using $K_{OutputKeyRef.keyNum}$ because the device receiving *SIGout* shares $K_{OutputKeyRef.keyNum}$ with the translating device. *OutputKeyRef.useChipId* = 0

	For generating variant key output signature: OutputKeyRef.keyNum = Slot number of the key to be used for generating the variant key. *SIGout* produced using a variant of $K_{OutputKeyRef.keyNum}$ because the device receiving *SIGout* shares a variant of $K_{OutputKeyRef.keyNum}$ with the translating device. OutputKeyRef.useChipId = 1 *OutputKeyRef.chipId* = ChipId of the device which receives SIG_{out} produced by a variant of $K_{OutputKeyRef.keyNum}$.
R_{E2}	External random value required for output signature generation. This will be the R from the destination of SIG_{out}. R_{E2} is obtained by calling the *Random* function on the device which will receive the SIG_{out} from the *Translate* function.

17.2.1 Input signature verification data format

This is the same format as used in the *Test* function. Refer to Section 16.2.1.

17.3 OUTPUT PARAMETERS

5   Table 266 describes each of the output parameters.

Table 266. Description of output parameters for Translate

Parameter	Description
ResultFlag	Indicates whether the function completed successfully or not. If it did not complete successfully, the reason for the failure is returned here. See Section 12.1.
R_{L2}	Local random value used in output signature (i.e SIG_{Out}).
SIG_{Out}	Output signature produced using *OutputKeyRef.keyNum* using the data format described in Figure 372. $SIG_{Out} = SIG_{OutKeyRef}(Data \mid R_{L2} \mid R_{E2})$.Refer to Section 10.1.3.1for details.

17.3.1 SIG_{out}

10   Figure 372 shows the data format for output signature generation from the *Translate* function.

17.4 FUNCTION SEQUENCE

The Translate command is illustrated by the following pseudocode:

```
Accept input parameters-InputKeyRef, DataLength
```

15

```
# Accept input parameter- Data based on DataLength


For i ← 0 to (DataLength - 1)
   Accept next Data
EndFor
```

Accept input parameters - R_E, SIG_E, OutputKeyRef, R_{E2}

```
        Check range of InputKeyRef.keyNum and OutputKeyRef.keyNum
5       If invalid
            ResultFlag ← Invalidkey
            Output ResultFlag
            Return
        EndIf
10      #Generate Signature
        SIG_L ← GenerateSignature(InputKeyRef,Data,R_E,R_L)  # Refer to Figure
        371.


        #Validate input signature
15      If(SIG_L = SIG_E)
            Update R_L to R_L2
        Else
            ResultFlag ← BadSig
            Output ResultFlag
20          Return
        EndIf


        #Generate output signature
        SIG_Out ← GenerateSignature(OutputKeyRef,Data,R_E,R_L)  # Refer to
25      Figure 372.
        Update R_L2 to R_L3
        ResultFlag ← Pass
        Output ResultFlag, R_L2, SIG_Out
        Return
30  18  WriteM1+
                Input:        VectNum, WordSelect, MVal
                Output:       ResultFlag
                Changes:      M_VectNum
                Availability: All devices
35
```

18.1 FUNCTION DESCRIPTION

The *WriteM1+* function is used to update selected words of M1+, subject to the permissions corresponding to those words stored in $P_{VectNum}$.

Note: Unlike WriteAuth, a signature is not required as an input to this function.

18.2 INPUT PARAMETERS

Table 267 describes each of the input parameters.

Table 267. Description of input parameters for WriteM1+

Parameter	Description
VectNum	Number of the memory vector to be written. Must be in range 1 to (NumVectors -1)
WordSelect	Selection of words to be written. 0- indicates corresponding word is not written. 1- indicates corresponding word is to be written as per input. If *WordSelect[N bit]* is set, then write to $M_{VectNum}$ word *N*.
MVal	Multiple of words corresponding to the number of words selected for write. Starts with LSW of $M_{VectNum}$.

Note: Since this function has no accompanying signatures, additional input parameter error checking is required.

18.3 OUTPUT PARAMETERS

Table 268 describes each of the output parameters.

Table 268. Description of output parameters for WriteM1+

Parameter	Description
ResultFlag	Indicates whether the function completed successfully or not. If it did not complete successfully, the reason for the failure is returned here. See Section 12.1.

18.4 FUNCTION SEQUENCE

The *WriteM1+* command is illustrated by the following pseudocode:

```
Accept input parameters VectNum, WordSelect


#Accept MVal as per WordSelect
MValTemp[16]  ← 0  # Temporary buffer to hold MVal after being read
For i ← 0 to MaxWordInM  # word 0 to word 15
```

```
        If(WordSelect[i] = 1)
            Accept next MVal
            MValTemp[i] ← MVal # Store MVal in temporary buffer
        EndIf
5   EndFor


    Check range of VectNum
    If invalid
        ResultFlag ← InvalidVector
10      Output ResultFlag
        Return
    EndIf


    #Checking non authenticated write permission for M1+
15

    PermOK ← CheckM1+Perm(VectNum,WordSelect)


    #Writing M with MVal

20  If(PermOK =1)
        WriteM(VectNum,MValTemp[])
        ResultFlag ← Pass
    Else
        ResultFlag ← InvalidPermission
25  EndIf
    Output ResultFlag
    Return
```

18.4.1   PermOK CheckM1+Perm (VectNum, WordSelect)

This function checks *WordSelect* against permission $P_{VectNum}$ for the selected word.

```
30  For i ← 0 to MaxWordInM # word 0 to word 15
        If(WordSelect[i] = 1) ∧ (P_VectNum[i] = 0) # Trying to write a
    ReadOnly word
            Return PermOK← 0
        EndIf
35  EndFor
    Return PermOK← 1
```

18.4.2   WriteM(VectNum, MValTemp[])

This function copies MValTemp to M$_{VectNum}$.

```
For i ←0 to MaxWordInM # Copying word from temp buff to M
    If(VectNum = 1)  # If M1
        P_VectNum[i]← 0 # Set permission to ReadOnly before writing
    EndIf
    M_VectNum[i]  ← MValTemp[i]                           # copy word
buffer to M word
    EndIf
EndFor
```

19   WriteFields

Input:	*FieldSelect, FieldVal*
Output:	*ResultFlag*
Changes:	*M$_{VectNum}$*
Availability:	*All devices*

19.1   FUNCTION DESCRIPTION

The *WriteFields* function is used to write new data to selected fields (stored in M0). The write is carried out subject to the non-authenticated write access permissions of the fields as stored in the appropriate words of M1 (see Section 8.1.1.3).

The *WriteFields* function is used whenever authorization for a write (i.e. a valid signature) is not required. The *WriteFieldsAuth* function is used to perform authenticated writes to fields. For example, decrementing the amount of ink in an ink cartridge field is permitted by anyone via the *WriteFields*, but incrementing it during a refill operation is only permitted using *WriteFieldsAuth*. Therefore *WriteFields* does not require a signature as one of its inputs.

19.2   INPUT PARAMETERS

Table 269 describes each of the input parameters.

Table 269. Description of input parameters for WriteFields

Parameter	Description
FieldSelect	Selection of fields to be written. 0- indicates corresponding field is not written. 1- indicates corresponding field is to be written as per input. If *FieldSelect [N bit]* is set, then write to Field *N* of M0.
FieldVal	Multiple of words corresponding to the words for all selected fields. Since Field0 starts at M0[15], *FieldVal* words starts with MSW of lower field.

Note: Since this function has no accompanying signatures, additional input parameter error checking is required especially if the QA Device communication channel has potential for error.

19.3   OUTPUT PARAMETERS

Table 270 describes each of the output parameters.

Table 270. Description of output parameters for WriteFields

Parameter	Description
ResultFlag	Indicates whether the function completed successfully or not. If it did not complete successfully, the reason for the failure is returned here. See Section 12.1.

19.4   FUNCTION SEQUENCE

The *WriteFields* command is illustrated by the following pseudocode:

```
        Accept input parameters FieldSelect


        #Accept FieldVal as per FieldSelect into a temporary buffer
        MValTemp


        #Find the size of each FieldNum to accept FieldData
        FieldSize[16] ← 0 # Array to hold FieldSize assuming there are 16
        fields
        NumFields← FindNumberOfFieldsInM0(M1,FieldSize)


        MValTemp[16] ← 0 # Temporary buffer to hold FieldVal after being
        read
        For i ← 0 to NumFields
           If FieldSelect[i] = 1
              If i = 0 # Check if field number is 0
                 PreviousFieldEndPos ← MaxWordInM
              Else
                 PreviousFieldEndPos ←M1[i-1].EndPos # position of the last
        word for the
                                                        # previous
        field
              EndIf
              For j ← (PreviousFieldEndPos -1) to M1[FieldNum].EndPos()
```

792

```
            MValTemp[j] = Next FieldVal word #Store FieldVal in
       MValTemp.
            EndFor
          EndIf
5      EndFor


       #Check non-authenticated write permissions for all fields in
       FieldSelect


10     PermOK ← CheckM0NonAuthPerm(FieldSelect,MValTemp,M0,M1)


       #Writing M0 with MValTemp if permissions allow writing


       If(PermOK =1)
15         WriteM(0,MValTemp)
           ResultFlag ← Pass
       Else
           ResultFlag ← InvalidPermission
       EndIf
20     Output ResultFlag
       Return
```

19.4.1   NumFields FindNumOfFieldsInM0(M1,FieldSize[])

This function returns the number of fields in M0 and an array FieldSize which stores the size of each field.

```
25     CurrPos ← 0
       NumFields ← 0
       FieldSize[16] ← 0 # Array storing field sizes


       For FieldNum ← 0 to MaxWordInM
30         If(CurrPos = 0) # check if last field has reached
               Return FieldNum #FieldNum indicates number of fields in M0
           EndIf
           FieldSize[FieldNum]← CurrPos - M1[FieldNum].EndPos
           If(FieldSize[FieldNum] < 0)
35             Error # Integrity problem with field attributes
```

```
                Return FieldNum # Lower M0 fields are still valid but higher
        M0 fields are

                                    # ignored

            Else

5               CurrPos← M1[FieldNum].EndPos

            EndIf

        EndFor
```

19.4.2   WordBitMapForField GetWordMapForField(FieldNum,M1)

This function returns the word bitmap corresponding to a field i.e the field consists of which
consecutive words.

```
        WordBitMapForField← 0

        WordMapTemp ← 0

        PreviousFieldEndPos ←M1[FieldNum -1].EndPos # position of the
        last word for the

                                                    # previous
        field

        For j ← (PreviousFieldEndPos +1) to M1[FieldNum].EndPos()
            # Set bit corresponding to the word position
            WordMapTemp ← SHIFTLEFT(1,j)

            WordBitMapForField ← WordMapTemp ∨ WordBitMapForField
        EndFor

        Return WordBitMapForField
```

19.4.3   PermOK CheckM0NonAuthPerm(FieldSelect,MValTemp[],M0,M1)

This functions checks non-authenticated write permissions for all fields in *FieldSelect*.

```
        PermOK CheckM0NonAuthPerm()

        FieldSize[16] ← 0

        NumFields ← FindNumOfFieldsInM0(FieldSize)

        # Loop through all fields in FieldSelect and check their
        # non-authenticated permission

        For i ← 0 to NumFields
            If FieldSelect[i] = 1 # check selected
                WordBitMapForField← GetWordMapForField(i,M1) #get word
        bitmap for field
                PermOK

            ← CheckFieldNonAuthPerm(i,WordBitMapForField,MValTemp,M0,)
```

```
                               # Check permission for field i in
        FieldSelect
             If(PermOK = 0)          #Writing is not allowed, return if
        permissions for field
                               # doesn't allow writing
            Return PermOK
          EndIf
        EndIf
      EndFor
      Return PermOK
```

19.4.4   PermOK

```
        CheckFieldNonAuthPerm(FieldNum,WordBitMapForField, MValTemp[],M0)
```

This function checks non authenticated write permissions for the field.

```
      DecrementOnly ← 0

      AuthRW ← M1[FieldNum].AuthRW

      NonAuthRW ← M1[FieldNum].AuthRW

      If(NonAuthRW = 0) # No NonAuth write allowed

         Return PermOK← 0

      EndIf

      If((AuthRW = 0)  ∧  (NonAuthRW = 1))# NonAuthRW allowed

         Return PermOK←1

      ElseIf(AuthRW = 1)  ∧  (NonAuthRW = 1)# NonAuth DecrementOnly
      allowed

         PermOK

      ← CheckInputDataForDecrementOnly(M0,MValTemp,WordBitMapForField)

         Return PermOK

      EndIf
```

19.4.5   PermOK CheckInputDataForDecrementOnly(M0,MValTemp[],WordBitMapForField)

This function checks the data to be written to the field is less than the current value.

```
      DecEncountered ← 0

      LessThanFlag ← 0

      EqualToFlag ← 0

      For i = MaxWordInM to 0

         If(WordBitMapForField[i] = 1) # starting word of the field -
      starting at MSW

            # comparing the word of temp buffer with M0 current value
```

```
        LessThanFlag ← M0[i] < MValTemp[i]

        EqualToFlag← M0[i] = MValTemp[i]

        # current value is less or previous value has been decremented

        If(LessThanFlag =1) ∨ (DecEncountered = 1)

            DecEncountered ← 1

            PermOK← 1

            Return PermOK

        ElseIf(EqualToFlag ≠1)   # Only if the value is greater than
        current and decrement not encountered in previous words

                PermOK← 0

                Return PermOK

        EndIf

    EndIf

EndFor
```

19.4.6   WriteM(VectNum, MValTemp[])

    Refer to Section 18.4.2 for details.

20   WriteFieldsAuth

        Input: *KeyRef, FieldSelect, FieldVal, R_E, SIG_E*

        Output: *ResultFlag*

        Changes: $_{M0}$ *and* R_L

        Availability: *All devices*

20.1   FUNCTION DESCRIPTION

The *WriteFieldsAuth* command is used to securely update a number of fields (in $_{M0}$). The write is carried out subject to the authenticated write access permissions of the fields as stored in the appropriate words of M1 (see Section 8.1.1.3). *WriteFieldsAuth* will either update all of the requested fields or none of them; the write only succeeds when *all* of the requested fields can be written to.

The *WriteFieldsAuth* function requires the data to be accompanied by an appropriate signature based on a key that has appropriate write permissions to the field, and the signature must also include the local R (i.e. nonce/challenge) as previously read from this QA Device via the *Random* function.

The appropriate signature can only be produced by knowing K_{KeyRef}. This can be achieved by a call to an appropriate command on a QA Device that holds a key matching K_{KeyRef}. Appropriate commands include *SignM, XferAmount, XferField, StartXfer*, and *StartRollBack*.

20.2   INPUT PARAMETERS

Table 271 describes each of the input parameters for WriteAuth.

Parameter	Description
KeyRef	For common key signature generation: KeyRef.keyNum = Slot number of the key to be used for testing the input signature. KeyRef.useChipId = 0
	No variant key signature generation required
FieldSelect	Selection of fields to be written. 0- indicates corresponding field is not written. 1- indicates corresponding field is to be written as per input. If FieldSelect [N bit] is set, then write to Field N of M0.
FieldVal	Multiple of words corresponding to the total number of words for all selected fields. Since Field0 starts at M0[15], FieldVal words starts with MSW of lower field.
RE	External random value used to verify input signature. This will be the R from the input signature generator (i.e device generating SIG_E).
SIGE	External signature required for authenticating input data. The external signature is either generated by a Translate or one of the Xfer functions. A correct SIG_E = SIG_{KeyRef}(data | R_E | R_L).

5   20.2.1   Input signature verification data format

Figure 373 shows the input signature verification data format for the WriteAuth function.

Table 272 gives the parameters included in SIG_E for Write Auth

Parameter	Length in bits	Value set internally	Value set from Input
RWSense	3	write constant = 001 Refer to Section 15.3.1.1	
FieldNum	4		●
ChipID	48	This QA Device's ChipId	
FieldData	32 per word		●
R_E	160		●
R_L	160	random value from device	

20.3 OUTPUT PARAMETERS

Table 273 describes each of the output parameters.

Table 273. Description of output parameters for WriteAuth

Parameter	Description
ResultFlag	Indicates whether the function completed successfully or not. If it did not complete successfully, the reason for the failure is returned here. See Section 12.1.

20.4 FUNCTION SEQUENCE

The *WriteAuth* command is illustrated by the following pseudocode:

```
Accept input parameters-KeyRef, FieldSelect,

#Accept FieldVal as per FieldSelect into a temporary buffer
MValTemp

#Find the size of each FieldNum to accept FieldData
FieldSize[16] ← 0 # Array to hold FieldSize assuming there are 16
fields
NumFields← FindNumberOfFieldsInM0(M1,FieldSize)

MValTemp[16] ← 0 # Temporary buffer to hold FieldVal after being
read
For i ← 0 to NumFields
    If i = 0 # Check if field number is 0
        PreviousFieldEndPos ← MaxWordInM
    Else
        PreviousFieldEndPos ←M1[i-1].EndPos # position of the last
word for the previous field
    EndIf
    For j ← (PreviousFieldEndPos -1) to M1[FieldNum].EndPos()
        MValTemp[j] = Next FieldVal word #Store FieldVal in MValTemp.
    EndFor
    EndIf
EndFor
```

```
        Accept R_E, SIG_E

        Check range of KeyRef.keyNum
        If invalid range
5          ResultFlag ← InvalidKey
           Output ResultFlag
           Return
        EndIf


10      #Generate message for passing to GenerateSignature function
        data ← (RWSense|FieldSelect|ChipId|FieldVal


        #Generate Signature
        SIG_L ← GenerateSignature(KeyRef,data,R_E,R_L) # Refer to Figure 373.
15
        #Check signature
        If(SIG_L = SIG_E)
           Update R_L to R_L2
        Else
20         ResultFlag ← BadSig
           Output ResultFlag
           Return
        EndIf


25

        #Check authenticated write permission for all fields in
        FieldSelect using KeyRef
        PermOK← CheckM0AuthPerm(FieldSelect,MValTemp,M0,M1,KeyRef)
        If(PermOK = 1)
30         WriteM(0,MValTemp[])# Copy temp buffer to M0
           ResultFlag ← Pass
        Else
           ResultFlag ← InvalidPermission
        EndIf
35      Output ResultFlag
        Return
```

20.4.1   PermOK CheckM0AuthPerm(FieldSelect,MValTemp[],M0, M1, KeyRef)

This functions checks non-authenticated write permissions for all fields in *FieldSelect* using KeyRef.

```
PermOK CheckM0NonAuthPerm()

FieldSize[16] ← 0

NumFields ← FindNumOfFieldsInM0(FieldSize)

# Loop through fields

For i ← 0 to NumFields

   If FieldSelect[i] = 1 # check selected

      WordBitMapForField← GetWordMapForField(i,M1) #get word
bitmap for field

         PermOK ← CheckAuthFieldPerm(i,WordBitMapForField,MValTemp,M0,
KeyRef)

               # Check permission for field i in FieldSelect

      If(PermOK = 0)        #Writing is not allowed, return if
            #permissions for field doesn't allow writing

         Return PermOK

      EndIf

   EndIf

EndFor

Return PermOK
```

20.4.2   PermOK CheckAuthFieldPerm(FieldNum, WordMapForField,MValTemp[],
         M0,KeyRef)

         This function checks authenticated permissions for an M0 field using KeyRef
         (whether KeyRef has write permissions to the field).

```
AuthRW ← M1[FieldNum].AuthRW

KeyNumAtt ← M1[FieldNum].KeyNum

If(AuthRW = 0) # Check whether any key has write permissions

   Return PermOK←0 # No authenticated write permissions

EndIf


# Check KeyRef has ReadWrite Permission to the field and it is
locked

If(KeyLock_KeyNum = locked)∧(KeyNumAtt = KeyRef.keyNum)

   Return PermOK← 1

Else # KeyNum is not a ReadWrite Key
```

```
        KeyPerms ← M1 [FieldNum] .DOForKeys # Isolate KeyPerms for
FieldNum


        # Check Decrement Only Permission for Key
        If (KeyPerms [KeyRef .keyNum] = 1) # Key is allowed to Decrement
field
            PermOK
    ← CheckInputDataForDecrementOnly (M0,MValTemp , WordMapForField)
        Else              # Key is a ReadOnly key
            PermOK←0
    EndIf
    EndIf
    Return PermOK
```

20.4.3   WordBitMapField GetWordMapForField(FieldNum,M1)

        Refer to Section 19.4.2 for details.

20.4.4   PermOK CheckInputDataForDecrementOnly(M0,MValTemp[],WordMapForField)

        Refer to Section 19.4.5 for details.

20.4.5   WriteM(VectNum, MValTemp[])

        Refer to Section 18.4.2 for details.

21   SetPerm

        Input: *VectNum, PermVal*

        Output: *ResultFlag, NewPerm*

        Changes: P_n

        Availability: All devices

21.1   FUNCTION DESCRIPTION

The *SetPerm* command is used to update the contents of $P_{VectNum}$ (which stores the permission for $M_{VectNum}$).

The new value for $P_{VectNum}$ is a combination of the old and new permissions in such a way that the more restrictive permission for each part of $P_{VectNum}$ is kept.

M0's permissions are set by M1 therefore they can't be changed.

M1's permissions cannot be changed by SetPerm. M1 is a write-once memory vector and its permissions are set by writing to it.

See Section 8.1.1.3 and Section 8.1.1.5 for more information about permissions.

21.2 INPUT PARAMETERS

Table 274 describes each of the input parameters for SetPerm.

Parameter	Description
VectNum	Number of the memory vector whose permission is being changed.
PermVal	Bitmap of permission for the corresponding Memory Vector.

Note: Since this function has no accompanying signatures, additional input parameter error checking is required.

21.3 OUTPUT PARAMETERS

Table 275 describes each of the output parameters for SetPerm.

Parameter	Description
ResultFlag	Indicates whether the function completed successfully or not. If it did not complete successfully, the reason for the failure is returned here. See Section 12.1.
Perm	If VectNum = 0, then no Perm is returned. If VectNum = 1, then old Perm is returned. If VectNum > 1, then new Perm is returned after $P_{VectNum}$ has been changed based on *PermVal*.

21.4 FUNCTION SEQUENCE

The *SetPerm* command is illustrated by the following pseudocode:

Accept input parameters- VectNum, PermVal

```
Check range of VectNum
If invalid
    ResultFlag ← InvalidVector
    Output ResultFlag
    Return
EndIf

If(VectNum = 0) # No permssions for M0
    ResultFlag ← Pass
```

802

```
        Output ResultFlag
        Return
    ElseIf(VectNum = 1)
        ResultFlag ← Pass
        Output ResultFlag
        Output P₁
        Return
    ElseIf(VectNum >1)
        # Check that only 'RW' parts are being changed
        # RW(1) → RO(0), RO(0) → RO(0), RW(1) → RW(1) - valid change
        # RO(0) → RW(1) - Invalid change
        # checking for change from ReadOnly to ReadWrite
        temp← ~P_VectNum ∧ PermVal
        If(temp = 1 )# If invalid change is 1
            ResultFlag ← InvalidPermission
            Output ResultFlag
        Else
            P_VectNum ← PermVal
            ResultFlag ← Pass
            Output ResultFlag
            Output P_VectNum
        EndIf
        Return
    EndIf
```

22   ReplaceKey

Input:	*KeyRef, KeyId, KeyLock, EncryptedKey,R_E, SIG_E*
Output:	*ResultFlag*
Changes:	$K_{KeyRef.keyNum}$ *and* R_L
Availability:	All devices

22.1   FUNCTION DESCRIPTION

The *ReplaceKey* command is used to replace the contents of a non-locked keyslot, which means replacing the key, its associated keyId, and the lock status bit for the keyslot. A key can only be replaced if the slot has not been locked i.e. the KeyLock for the slot is 0. The procedure for replacing a key also requires knowledge of the value of the current key in the keyslot i.e. you can only replace a key if you know the current key.

Whenever the *ReplaceKey* function is called, the caller has the ability to make this new key the final key for the slot. This is accomplished by passing in a new value for the KeyLock flag. A new KeyLock flag of 0 keeps the slot unlocked, and permits further replacements. A new KeyLock flag of 1 means the slot is now locked, with the new key as the final key for the slot i.e. no further key replacement is permitted for that slot.

5

22.2   INPUT PARAMETERS

Table 276 describes each of the input parameters for Replacekey.

Parameter	Description
KeyRef	*For common key signature generation: KeyRef.keyNum* = Slot number of the key to be used for testing the input signature, and will be replaced by the new key. *KeyRef.useChipId* = 0
	No variant key signature generation required
KeyId	KeyId of the new key. The LSB represents whether the new key is a variant or a common key.
KeyLock	Flag indicating whether the new key should be the final key for the slot or not. (1 = final key, 0 = not final key)
EncryptedKey	$SIG_{Kold}(R_E|R_L) \oplus K_{new}$ where K_{old} = KeyRef.getkey(). Refer to Section 10.1.3.1
RE	External random value required for verifying input signature. This will be the *R* from the input signature generator (device generating *SIG_E*). In this case the input signature is a generated by calling *the GetProgramKey* function on a *Key Programming device.*
SIGE	External signature required for authenticating input data and determining the new key from the *EncryptedKey.*

10

22.2.1   Input signature generation data format

Figure 374 shows the input signature generation data format for the *ReplaceKey* function.

Table 277 gives the parameters included in SIG_E for ReplaceKey.

Parameter	Length in bits	Value set internally	Value set from Input
ChipId	48	This QA Device's ChipId	
KeyId	32		●
R_E	160		●
EncryptedKey	160		●

22.3   OUTPUT PARAMETERS

Table 278 describes each of the output parameters for ReplaceKey.

Parameter	Description
ResultFlag	Indicates whether the function completed successfully or not. If it did not complete successfully, the reason for the failure is returned here. See Section 12.1.

22.4   FUNCTION SEQUENCE

The *ReplaceKey* command is illustrated by the following pseudocode:

```
Accept input parameters - KeyRef, KeyId, KeyLock, EncryptedKey, RE,
SIGE

Check KeyRef.keyNum range
If invalid
    ResultFlag ← InvalidKey
    Output ResultFlag
    Return
EndIf
#Generate message for passing to GenerateSignature function
data ← (ChipId|KeyId|KeyLock|RE|EncryptedKey)

#Generate Signature
```

805

```
SIG_L ← GenerateSignature(KeyRef,data,Null,Null) # Refer to Figure
374.


# Check if the key slot is unlocked

If(KeyLock # unlock)
    ResultFlag ← KeyAlreadyLocked
    Output ResultFlag
    Return
EndIf

#Test SIG_E
If (SIG_L # SIG_E)
    ResultFlag ← BadSig
    Output ResultFlag
    Return
EndIf
SIG_L ← GenerateSignature(Key,null,R_E,R_L)
    Advance R_L
    # Must be atomic - must not be possible to remove power and have
KeyId and KeyNum mismatched. Also preferable for KeyLock, although
not strictly  required.
    K_{KeyNum} ← SIG_L ⊕ EncryptedKey
    KeyId_{KeyNum} ← KeyId
    KeyLock_{KeyNum} ← KeyLock
    ResultFlag ← Pass
    Output ResultFlag
Return
```

23   SignM

Input:	KeyRef, FieldSelect, FieldValLength, FieldVal, ChipId, R_E
Output:	ResultFlag, R_L, SIG_{out}
Changes:	R_L
Availability:	Trusted device only

23.1   FUNCTION DESCRIPTION

The *SignM* function is used to generate the appropriate digital signature required for the authenticated write function *WriteFieldsAuth*. The *SignM* function is used whenever the caller wants to write a new value to a field that requires key-based write access.

The caller typically passes the new field value as input to the *SignM* function, together with the nonce (R_E) from the QA Device who will receive the generated signature. The *SignM* function then produces the appropriate signature *SIG_{out}*. Note that *SIG_{out}* may need to be translated via the *Translate* function on its way to the final *WriteFieldsAuth* QA Device.

The *SignM* function is typically used by the system to update preauthorisation fields (Section 31.4.3).

The key used to produce output signature *SIG_{out}* depends on whether the trusted device shares a common key or a variant key with the QA Device directly receiving the signature. The *KeyRef* object passed into the interface must be set appropriately to reflect this.

23.2   INPUT PARAMETERS

Table 279 describes each of the input parameters for SignM.

Parameter	Description
KeyRef	*For generating common key output signature:* *Ref.keyNum* = Slot number of the key for producing the output signature. SIG_{out} produced using $K_{KeyRef.keyNum}$ because the device receiving SIG_{out} shares $K_{KeyRef.keyNum}$ with the trusted device. *KeyRef.useChipId* = 0
	For generating variant key output signature: *KeyRef.keyNum* = Slot number of the key to be used for generating the variant key. SIG_{out} produced using a variant of $K_{KeyRef.keyNum}$ because the device receiving SIG_{out} shares a variant of $K_{KeyRef.keyNum}$ with the trusted device. *KeyRef.useChipId* = 1 *KeyRef.chipId* = ChipId of the device which receives SIG_{out}.
FieldNum	Field number of the field that will be written to.
FieldDataLength	The length of the FieldData in words.
FieldData	The value that will be written to the field selected by *FieldNum*.
R_E	External random value used in the output signature generation. R_E is obtained by calling the *Random* function on the device, which will receive the SIG_{out} from the *SignM* function, which in this case is the *WriteAuth* function or

	the *Translate* function.
ChipId	Chip identifier of the device whose *WriteAuth* function will be called subsequently to perform an authenticated write to its *FieldNum* of MO.

23.3 OUTPUT PARAMETERS

Table 280 describes each of the output parameters.

Table 280. Description of output parameters for SignM

Parameter	Description
ResultFlag	Indicates whether the function completed successfully or not. If it did not complete successfully, the reason for the failure is returned here. See Section 12.1.
R_L	Internal random value used in the output signature.
SIG$_{out}$	$SIG_{Out} = SIG_{KeyRef}(data \mid R_L \mid R_E)$ as shown in Figure 373. As per Figure 373, R_E is actually R_L and R_L is R_E with respect to device producing SIG_{out} to be applied to WriteAuth function.

23.3.1 SIG$_{out}$

Refer to Section 20.2.1.

23.4 FUNCTION SEQUENCE

The *SignM* command is illustrated by the following pseudocode:

```
Accept input parameters - KeyRef, FieldNum, FieldDataLength


# Accept FieldData words
For i = 0 to FieldValLength
    Accept next FieldData
EndFor


Accept ChipId, R_E


Check KeyRef.keyNum range
If invalid
    ResultFlag ← InvalidKey
    Output ResultFlag
    Return
EndIf
#Generate message for passing into the GenerateSignature function
```

```
data ← (RWSense|FieldSelect|ChipId|FieldVal)

#Generate Signature
SIG_out ← GenerateSignature(KeyRef,data,R_L,R_E)  # Refer to Section
20.2.1.
Advance R_L to R_L2
ResultFlag ← Pass
Output parameters ResultFlag, R_L,SIG_out
Return
```

FUNCTIONS ON A
KEY PROGRAMMING QA DEVICE

24 Concepts

The *key programming device* is used to replace keys in other devices.

The *key programming device* stores both the old key which will be replaced in the device being programmed, and the new key which will replace the old key in the device being programmed. The keys reside in normal key slots of the *key programming device*.

Any key stored in the *key programming device* can be used as an old key or a new key for the device being programmed, provided it is permitted by the *key replacement map* stored within the *key programming device*.

Figure 375 is representation of a *key replacement map*. The 1s indicates that the new key is permitted to replace the old key. The 0s indicates that key replacement is not permitted for those positions. The positions in Figure 13 which are blank indicate a 0.

According to the key replacement map in Figure 13, K_5 can replace K_1, K_6 can replace K_3, K_4, K_5,K_7, K_3 can replace K_2 , K_0 can replace K_2, and K_2 can replace K_6. No key can replace itself.

Figure 375._ Key replacement map

The *key replacement map* must be readable from an external system and must be updateable by an authenticated write. Therefore, the *key replacement map* must be stored in an MO field. This requires one of the keys residing in the *key programming device* to be have ReadWrite access to the *key replacement map*. This key is referred to as the *key replacement map key* and is used to update the *key replacement map*.

There will one key replacement map field in a key programming device.

No key replacement mappings are allowed to the *key replacement map key* because it should not be used in another device being programmed. To prevent the *key replacement map key* from being used in key replacement, in case the mapping has been accidentally changed, the *key replacement map key* is allocated a fixed key slot of 0 in all *key programming devices.* If a *GetProgram* function is invoked on the *key programming device* with the *key replacement map key* slot number 0 it immediately returns an error, even before the key replacement map is checked.

The keys K_0 to K_7 in the key programming device are initially set during the instantiation of the *key programming device*. Thereafter, any key can be replaced on the *key programming device* by another *key programming device* If a key in a key slot of the *key programming device* is being replaced, the *key replacement map* for the old key must be invalidated automatically. This is done by setting the row and column for the corresponding key slot to 0 For example, if K_1 is replaced, then column 1 and row 1 are set to 0, as indicated in Figure 376.

The new mapping information for K_1 is then entered by performing an authenticated write of the *key replacement map* field using the *key replacement map key*.

24.1 KEY REPLACEMENT MAP DATA STRUCTURE

As mentioned in Section 24, the *key replacement map* must be readable by external systems and must be updateable using an authenticated write by the *key replacement map key*. Therefore, the *key replacement map* is stored in an MO field of the *key programming device*. The map is 8×8 bits in size and therefore can be stored in a two word field. The LSW of *key replacement map* stores the mappings for K_0 - K_3.The MSW of *key replacement map* stores the mappings for K_4 - K_7. Referring to Figure 375, *key replacement map* LSW is 0x40092000 and MSW is 0x40224040. Referring to Figure 376, after K_1 is replaced in the *key programming device,* the value of the *key replacement map* LSW is 0x40090000 and MSW is 0x40224040.

The *key replacement map* field has an M1 word representing its attributes. The attribute setting for this field is specified in Table 281.

Table 281. Key replacement map attribute setting

Attribute name	Value	Explanation
Type	TYPE_KEY_MAP Refer to Appendix A.	Indicates that the field value represents a key replacement map. Only one such field per key programming QA Device.
KeyNum	0	Slot number of the *key replacement map key*.
NonAuthRW	0	No non authenticated writes is permitted.
AuthRW	1	Authenticated write is permitted.
KeyPerms	0	No Decrement Only permission for any key.
EndPos	Value such that field size is 2 words	

810

24.2   BASIC SCHEME

The Key Replacement sequence is shown Figure 377.

Following is a sequential description of the transfer and rollback process:

1.   The System gets a Random number from the QA Device whose keys are going to be replaced.

2.   The System makes a *GetProgramKey Request* to the Key Programming QA Device. The Key Programming QA Device must contain both keys for QA Device whose keys are being replaced - Old Keys which are the keys that exist currently (before key replacement), and the New Keys which are the keys which the QA Device will have after a successful processing of the ReaplceKey Request. The GetProgramKey Request is called with the Key number of the Old Key (in the Key Programming QA Device) and the Key Number of the New Key (in the Key Programming QA Device), and the Random number from (1). The Key Programming QA Device validates the *GetProgramKey Request* based on the *KeyReplacement map,* and then produces the necessary *GetProgramKey Output.* The *GetProgramKey Output* consists of the encrypted New Key (encryption done using the Old Key), along with a signature using the Old Key.

3.   The System then applies *GetProgramKey Output* to the QA Device whose key is being replaced, by calling the *ReplaceKey* function on it, passing in the *GetProgramKey Output.* The ReplaceKey function will decrypt the encrypted New Key using the Old Key, and then replace its Old Key with the decrypted New Key.

25   Functions

25.1   GETPROGAMKEY

Input:	*OldKeyRef, ChipId, R_E, KeyLock, NewKeyRef*
Output:	*ResultFlag, R_L,EncryptedKey, KeyIdOfNewKey, SIG_{out}*
Changes:	*R_L*
Availability:	*Key programming device*

25.1.1   Function description

The *GetProgramKey* works in conjunction with the *ReplaceKey* command, and is used to replace the specified key and its *KeyId.* This function is available on a *key programming device* and produces the necessary inputs for the *ReplaceKey* function. The *ReplaceKey* command is then run on the device whose key is being replaced.

The *key programming device* must have both the old key and the new key programmed as its keys, and the *key replacement map* stored in one of its MO field, before *GetProgramKey* can be called on the device.

Depending on the *OldKeyRef* object and the *NewKeyRef* object passed in, the *GetProgramKey* will produce a signature to replace a common key by a common key, a variant key by a common key, a common key by a variant key or a variant key by a variant key.

25.1.2 Input parameters

Table 282 describes each of the input parameters for GetProgramKey.

Parameter	Description
OldKeyRef	Old key is a common key: OldKeyRef.keyNum = Slot number of the old key in the Key Programming QA Device. The device whose key is being replaced, shares a common key $K_{OldKeyRef.keyNum}$ with the key programming device. OldKeyRef.useChipId = 0
	Old key is a variant key KeyRef.keyNum = Slot number of the old key in the Key Programming QA Device. that will be used to generate the variant key. The device whose key is being replaced, shares a variant of $K_{OldKeyRef.keyNum}$ with the key programming device. OldKeyRef.useChipId = 1 OldKeyRef.chipId = ChipId of the device whose variant of $K_{OldKeyRef.keyNum}$ key is being replaced.
ChipId	Chip identifier of the device whose key is being replaced.
RE	External random value which will be used in output signature generation. R_E is obtained by calling the Random function on the device being programmed. This will also receive the SIGout from the GetProgramKey function. SIGout is passed in to ReplaceKey function.
KeyLock	Flag indicating whether the new key should be unlocked/locked into its slot.
NewKeyRef	New key is a common key: NewKeyRef.keyNum = Slot number of the new key in the Key Programming QA Device. The device whose key is being replaced, will receive a common key $K_{NewKeyRef.keyNum}$ from the key pro gramming device. NewKeyRef.useChipId = 0
	NewKey is a variant key: NewKeyRef.keyNum = Slot number of the new key in the Key Programming QA Device. that will be used to generate the new variant key. The device whose key is being replaced, will receive a new key which is a variant of $K_{NewKeyRef.keyNum}$ from the key programming device. NewKeyRef.useChipId = 1 NewKeyRef.chipId = ChipId of the device receiving a new key, the new key is a variant of the $K_{NewKeyRef.keyNum}$.

5

25.1.3 Output parameters

Table 283 describes each of the output parameters for GetProgramKey.

Parameter	Description
ResultFlag	Indicates whether the function completed successfully or not. If it did not complete successfully, the reason for the failure is returned here. See Section 12.1 and Table 284
R_L	Internal random value used in the output signature.
EncryptedKey	$SIG_{Kold}(R_L|R_E) \oplus K_{new}$
KeyIdOfNewKey	KeyId of the new key.The LSB represents whether the new key is a variant or a common key.
SIG_{out}	$SIG_{out} = SIG_{Kold}(data \mid R_L \mid R_E)$

5

Table 284. ResultFlag definitions for GetProgramKey

Result Flag	Description
InvalidKeyReplacementMap	Key replacement map field invalid or doesn't exist.
KeyReplacementNotAllowed	Key replacement not allowed as per key replacement map.

25.1.3.1 SIG_{out}

10

Figure 378 shows the output signature generation data format for the GetProgramKey function.

25.1.4 Function sequence

The GetProgramKey command is illustrated by the following pseudocode:

```
Accept input parameters - OldKeyRef, ChipId, R_E, KeyLock,
NewKeyRef
```

15

```
-------------------------------------------------------------
# key replacement map key stored in K0, must not be used for key
replacement
```

20

```
If (OldKeyRef.keyNum = 0) ∨ (NewKeyRef.keyNum = 0)
    ResultFlag ← Fail
    Output ResultFlag
    Return
EndIf
```

25

```
-------------------------------------------------------------
```

```
          CheckRange(OldKeyRef.keyNum)
          If invalid
              ResultFlag ← InvalidKey
              Output ResultFlag
    5         Return
          EndIf
          ----------------------------------------------------------------
          CheckRange(NewKeyRef.keyNum)
          If invalid
   10         ResultFlag ← InvalidKey
              Output ResultFlag
              Return
          EndIf
          ----------------------------------------------------------------
   15     # Find M0 words that represent the key replacement map
          WordSelectForKeyMapField ←GetWordSelectForKeyMapField(M1)
          If(WordSelectForKeyMapField = 0)
              ResultFlag ← InvalidKeyReplacementMap
              Output ResultFlag
   20         Return
          EndIf
          ----------------------------------------------------------------
          #CheckMapPermits key replacement
          ReplaceOK
   25     ←CheckMapPermits(WordSelectForKeyMapField,OldKeyNum,NewKeyNum)
          If(ReplaceOK = 0)
              ResultFlag ← KeyReplacementNotAllowed
              Output ResultFlag
              Return
   30     EndIf
          ----------------------------------------------------------------
          #All checks are OK, now generate Signature with OldKey
```

SIG_L ← GenerateSignature(OldKeyRef,null,R_L,R_E)

#Get new key

35   K_{NewKey}← NewKeyRef.getKey()

#Generate Encrypted Key

814

```
EncryptedKey ← SIG_L ⊕ K_NewKey


#Set base key or variant key - bit 0 of KeyId
If(NewKeyRef.useChipId = 1)
    KeyId← 0x0001 ∧ 0x0001
Else
    KeyId ← 0x0001∧ 0x0000
EndIf


#Set the new key KeyId to the KeyId - bits 1-30 of KeyId
KeyIdOfNewKey←SHIFTLEFT(KeyIdOfNewKey,1)
KeyId← KeyId ∨ KeyIdOfNewKey


#Set the KeyLock as per input - bit 31 of KeyId
KeyLock← SHIFTLEFT(KeyLock,31)
#KeyId← KeyId ∨ KeyLock


#Generate message for passing in to the GenerateSignature function
data ← ChipId|KeyId|R_L|EncryptedKey


#Generate output signature
SIG_out ← GenerateSignature(OldKeyRef,data,null,null)
# Refer to Figure 378
Advance R_L to R_L2
ResultFlag ← Pass
Output ResultFlag, R_L,SIG_out,KeyId, EncryptedKey
Return
```

25.1.4.1 *WordSelectForField GetWordSelectForKeyMapField(M1)*

This function gets the words corresponding to the key replacement map in M0.

```
FieldSize[16] ← 0 # Array to hold FieldSize assuming there are 16
fields
NumFields ← FindNumberOfFieldsInM0(M1,FieldSize)


#Find the key replacement map field
For i ←0 to NumFields
    If(TYPE_KEY_MAP = M1[i].Type) # Field is key map field
```

```
        MapFieldNum ← i
        Return
    Endif
EndFor
```

5

```
#Get the words corresponding to the key replacement map
WordMapForField← GetWordMapForField(MapFieldNum,M1)
Return WordSelectForField
```

25.1.4.2 NumFields FindNumOfFieldsInM0(M1,FieldSize[])

10

Refer to Figure 19.4.1 for details

25.1.4.3 WordMapForField GetWordMapForField(FieldNum,M1)

Refer to Section 19.4.2 for details.

25.1.4.4 ReplaceOK CheckMapPermits(WordSelectForKeyMapField, OldKeyNum, NewKeyNum,M0)

15

This function checks whether key replacement map permits key replacement.

```
#Isolate KeyReplacementMap based on WordSelectForKeyMapField and M0
KeyReplacementMap[64 bit]
```

20

```
#Isolate permission bit corresponding for NewKeyNum in the map for
OldKeyNm
ReplaceOK ← KeyReplacementMap[(OldKeyNum × 8 + NewKeyNum) bit]
Return ReplaceOK
```

25.2   REPLACEKEY

25

Input: *KeyRef, KeyId, KeyLock, EncryptedKey,R_E, SIG_E*

Output: *ResultFlag*

Changes: *K_{KeyNum} and R_L*

Availability: *Key programming device*

25.2.1   Function description

30

This function is used for replacing a key in a key programming device and is similar to the generic *ReplaceKey* function(Refer to Section 24), with an additional step of setting the *KeyRef.keyNum* column and *KeyRef.keyNum* row *key replacement map* to 0.

25.2.2   Input parameters

Refer to Section 22.

35   25.2.3   Output parameters

Refer to Section 22.

25.2.4   Function sequence

The *ReplaceKey* command is illustrated by the following pseudocode:

```
Accept input parameters - KeyRef, KeyId, EncryptedKey,R_E, SIG_E



#Generate message for passing into GenerateSignature function
data ← (ChipId|KeyId|R_E|EncryptedKey)# Refer to Figure 374.
------------------------------------------------------------------

# Validate KeyRef, and then verify signature
ResultFlag = ValidateKeyRefAndSignature(KeyRef,data,R_E,R_L)
If (ResultFlag ≠ Pass)
   Output ResultFlag
   Return
EndIf
------------------------------------------------------------------

# Check if the key slot is unlocked
Isolate KeyLock for KeyRef
If(KeyLock = lock)
   ResultFlag ← KeyAlreadyLocked
   Output ResultFlag
   Return
EndIf
SIG_L ← GenerateSignature(Key,Null,R_E,R_L)
Advance R_L

# Find M0 words that represent the key replacement map
WordSelectForKeyMapField ←GetWordSelectForKeyMapField(M1)

# Set the bits corresponding to the KeyRef.keyNum row and column
to 0
# i.e invalidate the key replacement map for KeyRef.keyNum.
#Must be done before the key is replaced and must be atomic with
key replacement.
```

```
SetFlag
← SetKeyMapForKeyNum(WordSelectForKeyMapField,KeyRef.keyNum,M0)
If(SetFlag = 1)
```
 # Must be atomic - must not be possible to remove power and have
KeyId and

KeyNum mismatched

K_{KeyNum} ← SIG_L ⊕ *EncryptedKey*

$KeyId_{KeyNum}$ ← *KeyId*

```
KeyLockKeyNum ← KeyLock
ResultFlag ← Pass
Else
ResultFlag ← Fail
EndIf
Output ResultFlag
Return
```
25.2.4.1 *WordSelectForField GetWordSelectForKeyMapField(M1)*

Refer to Figure 25.1.4.1 for details.

25.2.4.2 *SetFlag SetKeyMapForKeyNum(WordSelectForKeyMapField,KeyNum, M0)*

This function invalidates the key replacement map for KeyNum.

#Isolate KeyReplacementMap based on WordSelectForKeyMapField and
M0
```
KeyReplacementMap [64 bit]
```

 # Set KeyNum row (all bits) to 0 in the KeyReplacementMap
```
For i = 0 to 7
    KeyReplacementMap[(KeyNum × 8 + i)bit] ← 0
EndFor
```

 # Set KeyNum column to 0 in the KeyReplacementMap
```
For i = 0 to 7
    KeyReplacementMap[(i×8 + KeyNum)bit] ← 0
EndFor
SetFlag ← 1
Return SetFlag
```

FUNCTIONS

UPGRADE DEVICE

(INK RE/FILL)

26   Concepts

5   26.1   PURPOSE

In a printing application, an ink cartridge contains an Ink QA Device storing the ink-remaining values for that ink cartridge. The ink-remaining values decrement as the ink cartridge is used to print.

When an ink cartridge is *physically* re/filled, the Ink QA Device needs to be *logically* re/filled as well. Therefore, the main purpose of an upgrade is to re/fill the ink-remaining values of an Ink QA Device

10   in an authorised manner.

The authorisation for a re/fill is achieved by using a *Value Upgrader QA Device which* contains all the necessary functions to re/write to the Ink QA Device. In this case, the value upgrader is called an *Ink Refill QA Device*, which is used to fill/refill ink amount in an Ink QA Device.

When an Ink Refill QA Device increases (additive) the amount of ink-remaining in an Ink QA Device,

15   the amount of ink-remaining in the Ink Refill QA Device is correspondingly decreased. This means that the Ink Refill QA Device can only pass on whatever ink-remaining value it itself has been issued with. Thus an Ink Refill QA Device can itself be replenished or topped up by another Ink Refill QA Device.

The Ink Refill QA Device can also be referred to as the Upgrading QA Device, and the Ink QA

20   Device can also be referred to as the QA Device being upgraded.

The refill of ink can also be referred to as a transfer of ink, or transfer of amount/valu, or an upgrade.

Typically, the logical transfer of ink is done only after a physical transfer of ink is successful.

26.2   REQUIREMENTS

25   The transfer process has two basic requirements:

- The transfer can only be performed if the transfer request is valid. The validity of the transfer request must be completely checked by the Ink Refill QA Device, before it produces the required output for the transfer. It must not be possible to apply the transfer output to the Ink QA Device, if the Ink Refill QA Device has been already been rolled back for that particular

30   transfer.

- A process of rollback is available if the transfer was not received by the Ink QA Device. A rollback is performed only if the rollback request is valid. The validity of the rollback request must be completely checked by the Ink Refill QA Device, before it adjusts its value to a previous value before the transfer request was issued. It must not be possible to rollback an

35   Ink Refill QA Device for a transfer which has already been applied to the Ink QA Device i.e the Ink Refill QA Device must only be rolled back for transfers that have actually failed.

26.3   BASIC SCHEME

The transfer and rollback process is shown in Figure 379.

Following is a sequential description of the transfer and rollback process:

1.   The System *Reads* the memory vectors M0 and M1 of the Ink QA Device. The output from the read which includes the M0 and M1 words of the Ink QA Device, and a signature, is passed as an input to the *Transfer Request*. It is essential that M0 and M1are read together. This ensures that the field information for M0 fields are correct, and have not been modified, or substituted from another device. Entire M0 and M1 must be read to verify the correctness of the subsequent *Transfer Request* by the Ink Refill QA Device.

2.   The System makes a *Transfer Request* to the Ink Refill QA Device with the amount that must be transferred, the field in the Ink Refill QA Device the amount must be transferred from, and the field in Ink QA Device the amount must be transferred to. The *Transfer Request* also includes the output from Read of the Ink QA Device. The Ink Refill QA Device validates the T*ransfer Request* based on the *Read* output, checks that it has enough value for a successful transfer, and then produces the necessary *Transfer Output*. The *Transfer Output* typically consists of new field data for the field being refilled or upgraded, additional field data required to ensure the correctness of the transfer/rollback, along with a signature.

3.   The System then applies the *Transfer Output* to the Ink QA Device, by calling an authenticated *Write* function on it, passing in the *Transfer Output*. The *Write* is either successful or not. If the *Write* is not successful, then the System will repeat calling the *Write* function using the same transfer output, which may be successful or not. If unsuccesful the System will initiate a *rollback* of the transfer. The *rollback* must be performed on the Ink Refill QA Device, so that it can adjust its value to a previous value before the current *Transfer Request* was initiated. It is not necessary to perform a rollback immediately after a failed *Transfer*. The Ink QA Device can still be used to print, if there is any ink remaining in it.

4.   The System starts a *rollback* by *Reading* the memory vectors M0 and M1 of the Ink QA Device.

5.   The System makes a *StartRollBack* Request to the Ink Refill QA Device with same input parameters as the *Transfer Request*, and the output from *Read* in (4). The Ink Refill QA Device validates the *StartRollBack* Request based on the *Read* output, and then produces the necessary *Pre-rollback output*. The *Pre-rollback output* consists only of additional field data along with a signature.

6.   The System then applies the *Pre-rollback* Output to the Ink QA Device, by calling an authenticated Write function on it, passing in the Pre-rollback output. The Write is either successful or not. If the Write is not successful, then either (6), or (5) and (6) must be repeated.

7.   The System then *Reads* the memory vectors M0 and M1 of the Ink QA Device.

8.   The System makes a *RollBack Request* to the Ink Refill QA Device with same input parameters as the *Transfer Request*, and the output from Read (7). The Ink Refill QA Device

validates the *RollBack Request* based on the *Read* output, and then rolls back its field corresponding to the transfer.

26.3.1   Transfer

As we mentioned, the Ink QA Device stores ink-remaining values in its M0 fields, and its corresponding M_1 words contains field information for its ink-remaining fields. The field information consists of the size of the field, the type of data stored in field and the access permission to the field. See Section 8.1.1 for details.

The Ink Refill QA Device also stores its ink-remaining values in its M0 fields, and its coressponding M_1 words contains field information for its ink-remaining fields.

26.3.1.1 Authorisation

The basic authorisation for a transfer comes from a *key*, which has authenticated ReadWrite permission (stored in field information as KeyNum) to the ink-remaining field (to which ink will be transferrred) in the Ink QA Device. We will refer to this key as the *refill key*. The *refill key* must also have authenticated decrement-only permission for the ink-remaining field (from which ink will be transferred) in the Ink Refill QA Device.

After validating the input transfer request, the Ink Refill QA Device will decrement the amount to be transferred from its ink-remaining field, and produce a transfer amount (previous ink-remaining amount in the Ink QA Device + transfer amount), additional field data, and a signature using the *refill key. Note that the Ink Refill QA Device can decrement its ink-remaining field only if the refill key has the permission to decrement it.*

The signature produced by the Ink Refill QA Device is subsequently applied to the Ink QA Device. The Ink QA Device will accept the transfer amount only if the signature is valid. *Note that the signature will only be valid if it was produced using the refill key which has write permission to the ink-remaining field being written.*

26.3.1.2 Data Type matching

The Ink Refill QA Device validates the transfer request by matching the *Type* of the data in ink-remaining information field of Ink QA Device to the *Type* of data in ink-remaining information field of the Ink Refill QA Device. This ensures that equivalent data Types are transferred i.e Network_OEM1_infrared ink is not transferred to Network_OEM1_cyan ink.

26.3.1.3 Addition validation

Additional validation of the transfer request must also be performed before a transfer output is generated by the Ink Refill QA Device. These are as follows:

• For the Ink Refill QA Device:

1. Whether the field being upgraded is actually present.

2. Whether the field being upgraded can hold the upgraded amount.

• For the Ink QA Device:

1. Whether the field from which the amount is transferred is actually present.

2. Whether the field has sufficient amount required for the transfer.

26.3.1.4 Rollback facilitation

To facilitate a rollback, the Ink Refill QA Device will store a *list of transfer requests* processed by it. This list is referred to as the *Xfer Entry* cache. Each record in the list consists of the transfer parameters corresponding to the transfer request.

26.3.2 Rollback

A rollback request is validated by looking through the Xfer Entry of the Ink Refill QA Device and finding the request that should be rolled back. After the right transfer request is found the Ink Refill QA Device checks that the output from the transfer request was not applied to the Ink QA Device by comparing the current Read of the Ink QA Device to the values in the Xfer Entry cache, and finally rolls back its ink-remaining field (from which the ink was transferred) to a previous value before the transfer request was issued.

The Ink Refill QA Device must be absolutely sure that the Ink QA Device didn't receive the transfer. This factor determines the additional fields that must be written along with transfer amount, and also the parameters of the transfer request that must be stored in the Xfer Entry cache to facilitate a rollback, to prove that the Printer QA Device didn't actually receive the transfer.

26.3.2.1 Sequence fields

The rollback process must ensure that the transfer output (which was previously produced) for which the rollback is being performed, cannot be applied after the rollback has been performed. How do we achieve this? There are two separate decrement-only *sequence* fields (*SEQ_1* and *SEQ_2*) in the Ink QA Device which can only be decremented by the Ink Refill QA Device using the *refill key*. The nature of data to be written to the sequence fields is such that either the transfer output or the pre-rollback output can be applied to the Ink QA Device, but not both i.e they must be mutually exclusive.Refer to Table 285 for details.

Table 285. Sequence field data for Transfer and Pre-rollback

Function	Sequence Field data written to Ink QA Device		Explanation
	SEQ_1	SEQ_2	
Initialised	0xFFFFFFFF	0xFFFFFFFF	Written using the *sequence* key which is different from the refill key
Write using Transfer Output	(Previous Value - 2) If Previous Value =intialised value then 0xFFFFFFFD	(Previous Value - 1) If Previous Value = intialised value then 0xFFFFFFFE	Written using the *refill* key using the refill key which has decrement-only permission on the fields. *Value cannot be written if pre-*

			rollback output is already written.
Write usiing Pre-rollback	(Previous Value - 1) If Previous Value =intialised value then 0xFFFFFFFE	(Previous Value - 2) If Previous Value = intialised value then 0xFFFFFFFD	Written using the refill key using the refill key which has decrement-only permissionon the fields. *Value can be written only if Transfer Output has not been written.*

The two sequence fields are initialised to 0xFFFFFFFF using *sequence key*. The sequence key is different to the refill key, and has authenticated ReadWrite permission to both the sequence fields. The transfer output consists of the new data for the field being upgraded, field data of the two sequence fields, and a signature using the refill key. The field data for SEQ_1 is decremented by 2 from the original value that was passed in with the transfer request. The field data for SEQ_2 is decremented by 1 from the original value that was passed in with the transfer request.

The pre-rollback output consists only of the field data of the two sequence fields, and a signature using the refill key. The field data for SEQ_1 is decremented by 1 from the original value that was passed in with the transfer request. The field data for SEQ_2 is decremented by 2 from the original value that was passed in with the transfer request.

Since the two sequence fields are decrement-only fields, the writing of the transfer output to QA Device being upgraded will prevent the writing of the pre-rollback output to QA Device being upgraded. If the writing of the transfer output fails, then pre-rollback can be written. However, the transfer output cannot be written after the pre-rollback has been written.

Before a rollback is performed, the Ink Refill QA Device must confirm that the sequence fields was successfully written to the pre-rollback values in the Ink QA Device. Because the sequence fields are Decrement-Only fields, the Ink QA Device will allow pre-rollback output to be written only if the upgrade output has not been written. It also means that the transfer output cannot be written after the pre-rollback values have been written.

26.3.2.1.1   Field information of the sequence data field

For a device to be upgradeable the device must have two sequence fields SEQ_1 and SEQ_2 which are written with sequence data during the *transfer sequence*. Thus all upgrading QA devices, ink QA Devices and printer QA Devices must have two *sequence* fields. The upgrading QA Devices must also have these fields because they can be upgraded as well.

The *sequence* field information is defined in Table 286.

Table 286. Sequence field information

Attribute Name	Value	Explanation
Type	TYPE_SEQ_1 or TYPE_SEQ_2.	See Appendix A for exact value.
KeyNum	Slot number of the *sequence key*.	*Only the sequence key has authenticated ReadWrite access to this field.*
Non Auth RW Perm	0	Non authenticated ReadWrite is not allowed to the field.
Auth RW Perm	1	Authenticated (key based) ReadWrite access is allowed to the field.
KeyPerm	KeyPerms[KeyNum] = 0	KeyNum is the slot number of the *sequence* key, which has *ReadWrite* permission to the field.
	KeyPerms[Slot number of the refill key] = 1	Refill key can *decrement* the sequence field.
	KeyPerms[others= 0 ..7(except refill key)] = 0	All other keys have *ReadOnly* access.
End Pos		Set as required. Size is typically 1 word.

26.3.3   Upgrade states

There are three states in an transfer sequence, the first state is initiated for every transfer, while the next two states are initiated only when the transfer fails. The states are - Xfer, StartRollback, and
5   Rollback.

26.3.3.1   Upgrade Flow

Figure 380 shows a typical upgrade flow.

26.3.3.2   Xfer

This state indicates the start of the transfer process, and is the only state required if the transfer is
10   successful. During this state, the Ink Refill QA Device adds a new record to its Xfer Entry cache, decrements its amount, produces new amount, new sequence data (as described in Section 26.3.2.1) and a signature based on the refill key.

The Ink QA Device will subsequently write the new amount and new sequence data, after verifying the signature. If the new amount can be successfully written to the Ink QA Device, then this will
15   finish a successful transfer.

If the writing of the new amount is unsuccessful (result returned is BAD SIG), the System will re-transmit the transfer output to the Ink QA Device, by calling the authenticated Write function on it again, using the same transfer output.

824

If retrying to write the same transfer output fails repeatedly, the System will start the rollback process on Ink Refill QA Device, by calling the Read function on the Ink QA Device, and subsequently calling the StartRollBack function on the Ink Refill QA Device. After a successful rollback is performed, the System will invoke the transfer sequence again.

5   *26.3.3.3 StartRollBack*

This state indicates the start of the rollback process. During this state, the Ink Refill QA Device produces the next sequence data and a signature based on the *refill key*. This is also called a pre-rollback, as described in Section 26.3.2.

The pre-rollback output can only be written to the Ink QA Device, if the previous transfer output has

10   not been written. The writing of the *pre-rollback* sequence data also ensures, that if the previous transfer output was captured and not applied, then it cannot be applied to the Ink QA Device in the future.

If the writing of the pre-rollback output is unsuccessful (result returned is BAD SIG), the System will re-transmit the pre-rollback output to the Ink QA Device, by calling the authenticated Write function

15   on it again, using the same pre-rollback output.

If retrying to write the same pre-rollback output fails repeatedly, the System will call the StartRollback on the Ink Refill QA Device again, and subsequently calling the authenticated Write function on the Ink QA Device using this output.

26.3.3.4 Rollback

20   This state indicates a successful deletion (completion) of a transfer sequence. During this state, the Ink Refill QA Device verifies the sequence data produced from StartRollBack has been correctly written to Ink Refill QA Device, then rolls its ink-remaining field to a previous value before the transfer request was issued.

26.3.4   Xfer Entry cache

25   The *Xfer Entry* data structure must allow for the following:

- Stores the *transfer state* and *sequence* data for a given *transfer sequence*.

- Store all data corresponding to a given transfer, to facilitate a *rollback* to the previous value before the transfer output was generated.

The *Xfer Entry cache* depth will depend on the QA Chip Logical Interface implementation. For some

30   implementations a single *Xfer Entry* value will be saved. If the Ink Refill QA Device has no powersafe storage of *Xfer Entry cache,* a power down will cause the erasure of the *Xfer Entry cache* and the Ink Refill QA Device will not be able to *rollback* to a pre-power-down value.

A dataset in the *Xfer Entry cache* will consist of the following:

- Information about the QA Device being upgraded:

35   a.   ChipId of the device.

b.   FieldNum of the M0 field (i.e what was being upgraded).

- Information about the upgrading QA Device:

a. FieldNum of the M0 field used to transfer the amount from.

• XferVal - the transfer amount.

• Xfer State- indicating at which state the *transfer sequence* is. This will consist of:

a. State definition which could be one of the following: - Xfer,

5   StartRollBack and complete/deleted.

b. The value of sequence data fields SEQ_1 and SEQ_2.

26.3.4.1 Adding new dataset

A new dataset is added to Xfer Entry cache by the Xfer function.

There are three methods which can be used to add new dataset to the *Xfer Entry* cache. The

10   methods have been listed below in the order of their priority:

1. *Replacing existing dataset in Xfer Entry cache with new dataset based on ChipId and FieldNum of the Ink QA Device in the new dataset.* A matching ChipId and FieldNum could be found because a previous *transfer* output corresponding to the dataset stored in the *Xfer Entry* cache has been correctly received and processed by the Ink Refill QA Device, and a

15   new transfer request for the same Ink QA Device, same field, has come through to the Ink Refill QA Device.

2. *Replace existing dataset cache with new dataset based on the Xfer State.* If the Xfer State for a dataset indicates deleted (complete), then such a dataset will not be used for any further functions, and can be overwritten by a new dataset.

20   3. *Add new dataset to the end of the cache.* This will automatically delete the oldest dataset from the cache regardless of the *Xfer State*.

26.4 DIFFERENT TYPES OF TRANSFER

There can be three types of transfer:

• *Peer to Peer Transfer* - This transfer could be one of the 2 types described below:

25   a. From an Ink Refill QA Device to a Ink QA Device. This is performed when the Ink QA Device is refilled by the Ink Refill QA Device.

b. From one Ink Refill QA Device to another Ink Refill QA Device, where both QA Devices belong to the same OEM. This is typically performed when OEM divides ink from one Ink Refill QA Device to another Ink Refill QA Device, where both devices belong to the same

30   OEM

• *Heirachical Transfer*- This is a transfer from one Ink Refill QA Device to another Ink Refill QA Device, where the QA Devices belong to different organisation, say ComCo and OEM. This is typically performed when ComCo divides ink from its refill device to several refill devices belonging to several OEMs.

35   Figure 381 is a representation of various authorised ink refill paths in the printing system.

26.4.1 Hierarchical transfer

Referring to Figure 381, this transfer is typically performed when ink is transferred from ComCo's Ink Refill QA Device to OEM's Ink Refill QA Device, or from QACo's Ink Refill QA Device to ComCo's Ink Refill QA Device.

26.4.1.1 Keys and access permission

5   We will explain this using a transfer from ComCo to OEM.

There is an *ink-remaining* field associated with the ComCo's Ink Refill QA Device. This *ink-remaining* field has *two keys* associated with:

- The *first key* transfers ink to the device from another refill device (which is higher in the heirachy), fills/refills (upgrades) the device itself. This key has authenticated ReadWrite
10   permission to the field.

- The *second key* transfers ink from it to other devices (which are lower in the heirachy), fills/refills (upgrades) other devices from it. This key has authenticated decrement-only permission to the field.

There is an *ink-remaining* field associated with the OEM's Ink refill device. This *ink-remaining* field
15   has *a single key* associated with:

- This *key* transfers ink to the device from another refill device (which is higher or at the same level in the hierarchy), fills/refills (upgrades) the device itself, and additionally transfers ink from it to other devices (which are lower in the heirachy), fills/refills (upgrades) other devices from it. Therefore, this key has both authenticated ReadWrite and decrement-only permission to the field.

20   For a successful transfer ink from ComCo's refill device to an OEM's refill device, the ComCo's refill device and the OEM's refill device must share a *common* key or a *variant* key. This key is *fill/refill key* with respect to the OEM's refill device and it is the *transfer key* with respect to the ComCo's refill device.

For a ComCo to successfully fill/refill its refill device from another refill device (which is higher in the
25   heirachy possibly belonging to the QACo), the ComCo's refill device and the QACo's refill device must share a *common* key or a *variant* key. This key is *fill/refill key* with respect to the ComCo's refill device and it is the *transfer key* with respect to the QACo's refill device.

26.4.1.1.1 Ink - remaining field information

Table 287 shows the field information for an $_{M0}$ field storing logical ink-remaining amounts in the
30   refill device and which has the ability to transfer down the heirachy.

Attribute Name	Value	Explanation
Type	For e.g - TYPE_HIGHQUALITY_BLACK_INK[a]	Type describing the logical ink stored in the ink-remaining field in the refill device.
KeyNum	Slot number of the *refill* key.	Only the *refill key* has authenticated *ReadWrite* access to this field.

Non Auth RW Perm[b]	0	Non authenticated ReadWrite is not allowed to the field.
Auth RW Perm[c]	1	Authenticated (key based) ReadWrite access is allowed to the field.
KeyPerm	KeyPerms[KeyNum] = 0	KeyNum is the slot number of the refill key, which has ReadWrite permission to the field.
	KeyPerms[Slot Num of transfer key] = 1	Transfer key can decrement the field.
	KeyPerms[others= 0..7(except transfer key)] = 0	All other keys have ReadOnly access.
End Pos	Set as required.	Depends on the amount of logical ink the device can store and storage resolution - i.e in picolitres or in microlitres.

a. This is a sample type only and is not included in the Type Map in Appendix A.

b. Non authenticated Read Write permission.

c. Authenticated Read Write permission.

5   26.4.2   Peer to Peer transfer

Referring to Figure 381, this transfer is typically performed when ink is transferred from OEM's Ink Refill Device to another Ink Refill Device belonging to the same OEM, or OEM's Ink Refill Device to Ink Device belonging to the same OEM.

26.4.2.1 Keys and access permission

10   There is an *ink-remaining* field associated with the refill device which transfers ink amounts to other refill devices (peer devices), or to other ink devices. This *ink-remaining* field has *a single key* associated with:

- This *key* transfers ink to the device from another refill device (which is higher or at the same level in the heirachy), fills/refills (upgrades) the device itself, and additionally transfers ink

15   from it to other devices (which are lower in the heirachy), fills/refills (upgrades) other devices from it.

This key is referred to as the *fill/refill key* and is used for *both fill/refill and transfer*. Hence, this *key* has both *ReadWrite* and *Decrement-Only* permission to the *ink-remaining* field in the *refill device*.

26.4.2.1.1   Ink-remaining field information

20   Table 288 shows the field information for an $_{M0}$ field storing logical ink-remaining amounts in the refill device with the ability to transfer between peers.

Attribute Name	Value	Explanation
Type	For e.g - TYPE_HIGHQUALITY_BLACK_INK[a]	Type describing the logical ink stored in the ink-remaining field in the refill device.
KeyNum	Slot number of the *refill* key.	Only the *refill key* has authenticated *ReadWrite* access to this field.
Non Auth RW Perm[b]	0	*Non authenticated ReadWrite* is not allowed to the field.
Auth RW Perm[c]	1	*Authenticated (key based) ReadWrite access* is allowed to the field.
KeyPerm	KeyPerms[KeyNum] = 1	KeyNum is the slot number of the *refill* key, which has *ReadWrite* and *Decrement* permission to the field.
	KeyPerms[others= 0 ..7(except KeyNum)] = 0	All other keys have *ReadOnly* access.
End Pos	Set as required.	Depends on the amount of logical ink the device can store and storage resolution - i.e in picolitres or in microlitres.

a. This is a sample type only and is not included in the Type Map in Appendix A.

b. Non authenticated Read Write permission.

5   c. Authenticated Read Write permission.

27   Functions

27.1   XFERAMOUNT

> *Input:*   *KeyRef, $_{M0}$OfExternal, $_{M1}$OfExternal, ChipId, FieldNumL,*
>
> *FieldNumE, XferValLength, XferVal, InputParameterCheck*
>
> 10   *(optional), R_E, SIG$_E$, R_{E2}*
>
> *Output:*   *ResultFlag, FieldSelect, FieldVal, R_{L2}, SIG$_{out}$*
>
> *Changes:*   *$_{M0}$ and R_L*
>
> *Availability*   *Ink refill QA Device*

27.1.1   Function description

15   The *XferAmount* function produces data and signature for updating a given $_{M0}$ field. This data and signature when applied to the appropriate device through the *WriteFieldsAuth* function, will update the $_{M0}$ field of the device.

The system calls the *XferAmount* function on the upgrade device with a certain *XferVal*, this *XferVal* is validated by the *XferAmount* function for various rules as described in Section 27.1.4, the function then produces the data and signature for the passing into the *WriteFieldsAuth* function for the device being upgraded.

5   The transfer amount output consists of the new data for the field being upgraded, field data of the two sequence fields, and a signature using the refill key. When a transfer output is produced, the sequence field data in *SEQ_1* is decremented by 2 from the previous value*(as passed in with the input)*, and the sequence field data in *SEQ_2* is decremented by 1 from the previous value *(as passed in with the input)*.

10   Additional *InputParameterCheck* value must be provided for the parameters not included in the SIG_E, if the transmission between the System and Ink Refill QA Device is error prone, and these errors are not corrected by the transimission protocol itself. *InputParameterCheck* is *SHA-1[FieldNumL | FieldNumE | XferValLength | XferVal]*, and is required to ensure the integrity of these parameters, when these inputs are received by the Ink Refill QA Device. This will prevent an

15   incorrect transfer amount being deducted.

The *XferAmount* function must first calculate the *SHA-1[FieldNumL | FieldNumE | XferValLength | XferVal]*, compare the calculated value to the value received (*InputParameterCheck*) and only if the values match act upon the inputs.

27.1.2   Input parameters

20   Table 289 describes each of the input parameters for XferAmount function.

Parameter	Description
KeyRef	*For comsmon key input and output signature: KeyRef.keyNum* = Slot number of the key to be used for testing input signature and producing the output signature. SIG_E produced using $K_{KeyRef.keyNum}$ by the QA Device being upgraded. *SIGout* produced using $K_{KeyRef.keyNum}$ for delivery to the QA Device being upgraded. *KeyRef.useChipId* = 0
	For variant key input and output signatures: KeyRef.keyNum = Slot number of the key to be used for generating the variant key. SIG_E produced using a variant of $K_{KeyRef.keyNum}$ by the QA Device being upgraded. *SIGout* produced using a variant of $K_{KeyRef.keyNum}$ for delivery to the QA Device being upgraded. *KeyRef.useChipId* = 1 *KeyRef.chipId* = ChipId of the device which generated SIG_E and will receive *SIGout*.
$_{M0}$*OfExternal*	All 16 words of $_{M0}$ of the QA Device being upgraded.
$_{M1}$*OfExternal*	All 16 words of $_{M1}$ of the QA Device being upgraded.
ChipId	ChipId of the QA Device being upgraded.

FieldNumL	M_0 field number of the local (refill) device from which the value will be transferred.
FieldNumE	M_0 field number of the QA Device being upgraded to which the value will be transferred.
XferValLength	XferVal length in words.Non zero length required.
XferVal	The logical amount that will be transferred from the local device to the external device.
R_E	External random value used to verify input signature. This will be the R from the input signature generator (i.e device generating SIG_E). The input signal generator in this case, is the device being upgraded or a translation device.
R_{E2}	External random value used to produce output signature. This will be R obtained by calling the Random function on the device which will receive the SIG_{out} from the XferAmount function. The device receiving the SIG_{out} in this case, is the device being upgraded or a translation device.
SIG_E	External signature required for authenticating input data. The input data in this case, is the output from the Read function performed on the device being upgraded. A correct SIG_E = SIG_{KeyRef}(Data | R_E | R_L).

27.1.2.1 Input signature verification data format

The input signature passed in to the *XferAmount* function is the output signature from the *Read* function of the *Ink QA Device*.

5   Figure 382 shows the input signature verification data format for the *XferAmount* function.

Table 290 gives the parameters included in SIG_E for XferAmount.

Parameter	Length in bits	Value set internally	Value set from Input
RWSense	3	000 Refer to Section 15.3.1.1	
MSelect	4	0011	
KeyIdSelect	8	00000000	
ChipId	48		ChipId of the QA Device being upgraded
WordSelect for M_0	16	All bits set to 1	
WordSelect for M_1	16	All bits set to 1	

M0	512		●
M1	512		●
R_E	160		●
R_L	160	Based on the internal R	●

The *XferAmount* function is not passed all the parameters required to generate SIG_E. For producing SIG_L which is used to test SIG_E, the function uses the expected values of some the parameters.

27.1.3   Output parameters

Table 291 describes each of the output parameters for XferAmount.

Parameter	Description
ResultFlag	Indicates whether the function completed successfully or not. If it did not complete successfully, the reason for the failure is returned here. See Table 47.
FieldSelect	Selection of fields to be written In this case the bit corresponding to *SEQ_1* , *SEQ_2* and to *FieldNumE* are set to 1. All other bits are set to 0.
FieldVal	Updated data words for *Sequence data* field *and FieldNumE* for QA Device being upgraded. Starts with LSW of lower field. This must be passed as input to the *WriteFieldsAuth* function of the QA Device being upgraded.
R_{L2}	Internal random value required to generate output signature. This must be passed as input to the *WriteFieldsAuth* function or *Translate* function of the QA Device being upgraded.
SIG_{out}	Output signature which must be passed as an input to the *WriteFieldsAuth* function of the QA Device being upgraded. $SIG_{out} = SIG_{KeyRef}(data \mid R_{L2} \mid R_{E2})$ as per Figure 373.

Table 292. Result Flag definitions for XferAmount

ResultFlag Definition	Description
FieldNumEInvalid	FieldNum to which the amount is being transferred, or which is being upgraded in the QA Device being upgraded is invalid.
SeqFieldInvalid	The sequence field for the QA Device being upgraded is invalid.
FieldNumEWritePermInvalid	FieldNum to which the amount is being transferred, or which is being upgraded in the QA Device being upgraded has no authenticated write permission.
FieldNumLInvalid	FieldNum from which the amount is being transferred, or from which the value is being copied in the Upgrading QA Device is invalid.
FieldNumLWritePermInvalid	FieldNum from which the amount is being transferred in the Upgrading QA Device has no authenticated permission, or no authenticated permission with the KeyRef.
TypeMismatch	Type of the data from which the amount is being transferred in the Upgrading QA Device, doesn't match the Type of data to which the amount in being transferred in the Device being upgraded.
UpgradeFieldEInvalid	Only applicable for transferring count-remaining values. The upgrade field associated with the count-remaining field in the QA Device being upgraded is invalid.
UpgradeFieldLInvalid	Only applicable for transferring count-remaining values. The upgrade field associated with the count-remaining field in the Upgrading QA Device is invalid.
UpgradeFieldMismatch	Only applicable for transferring count-remaining values. Type of the data in the upgrade field in the Upgrading QA Device, doesn't match the Type of data in the upgrade field in the Device being upgraded.
FieldNumESizeInsufficient	FieldNum to which the amount is being transferred, or which is being upgraded in the QA Device is not big enough to store the transferred data.
FieldNumLAmountInsufficient	FieldNum in the Upgrading QA Device from which the amount is being transferred doesn't have the amount required for the transfer.

27.1.3.1 SIG$_{Out}$

Refer to Section 20.2.1 for details.

27.1.4 Function sequence

The *XferAmount* command is illustrated by the following pseudocode:

```
        Accept input parameters-KeyRef, M0OfExternal, M1OfExternal,
        ChipId, FieldNumL, FieldNumE, XferValLength


        # Accept XferVal words
   5    For i ← 0 to XferValLength
           Accept next XferVal
        EndFor


        Accept R_E, SIG_E, R_E2
  10    #Generate message for passing into ValidateKeyRefAndSignature
        function
        data ← (RWSense|MSelect|KeyIdSelect|ChipId|WordSelect|M0|M1)
                 # Refer to Figure 382.


  15    -------------------------------------------------------------------


        # Validate KeyRef, and then verify signature
        ResultFlag = ValidateKeyRefAndSignature(KeyRef,data,R_E,R_L)
        If (ResultFlag ≠ Pass)
  20       Output ResultFlag
           Return
        EndIf
        -------------------------------------------------------------------


  25    #Validate FieldNumE
        # FieldNumE is present in the device being upgraded
        PresentFlagFieldNumE ← GetFieldPresent(M1OfExternal,FieldNumE)


        # Check FieldNumE present flag
  30    If(PresentFlagFieldNumE ≠ 1)
           ResultFlag ← FieldNumEInvalid
           Output ResultFlag
           Return
        EndIf
  35
        -------------------------------------------------------------------
        ----------------
```

834

```
        # Check Seq Fields Exist and get their Field Num
        # Get Seqdata field SEQ_1 num for the device being upgraded
        XferSEQ_1FieldNum← GetFieldNum(M1OfExternal, SEQ_1)

5

        # Check if the Seqdata field SEQ_1 is valid
        If(XferSEQ_1FieldNum invalid)
            ResultFlag ← SeqFieldInvalid
10          Output ResultFlag
            Return
        EndIf
        # Get Seqdata field SEQ_2 num for the device being upgraded
        XferSEQ_2FieldNum← GetFieldNum(M1OfExternal, SEQ_2)
15

        # Check if the Seqdata field SEQ_2 is valid
        If(XferSEQ_2FieldNum invalid)
            ResultFlag ← SeqFieldInvalid
            Output ResultFlag
20          Return
        EndIf

        ------------------------------------------------------------------

     #Check write permission for FieldNumE
25          PermOKFieldNumE ← CheckFieldNumEPerm(M1OfExternal,FieldNumE)
            If(PermOKFieldNumE ≠ 1)
                ResultFlag ← FieldNumEWritePermInvalid
                Output ResultFlag
                Return
30          EndIf
        ------------------------------------------------------------------
        #Check that both SeqData fields have Decrement-Only permission
        with the same key
        #that has write permission on FieldNumE
35          PermOKXferSeqData ← CheckSeqDataFieldPerms(M1OfExternal,
                                    XferSEQ_1FieldNum,
        XferSEQ_2FieldNum,FieldNumE)
```

835

```
        If(PermOKXferSeqData ≠ 1)

            ResultFlag ← SeqWritePermInvalid

            Output ResultFlag

            Return
  5     EndIf
        -----------------------------------------------------------------

        # Get SeqData SEQ_1 data from device being upgraded
        GetFieldDataWords(XferSEQ_1FieldNum,


 10         XferSEQ_1DataFromDevice,M0OfExternal,M1OfExternal)


        # Get SeqData SEQ_2 data from device being upgraded
        GetFieldDataWords(XferSEQ_2FieldNum,
                          XferSEQ_2DataFromDevice,
 15     M0OfExternal,M1OfExternal)


        -----------------------------------------------------------------

        # FieldNumL is a present in the refill device
        PresentFlagFieldNumL ← GetFieldPresent(M1,FieldNumL)
 20     If(PresentFlagFieldNumL ≠ 1)

            ResultFlag ← FieldNumLInvalid

            Output ResultFlag

            Return
        EndIf
 25

        #Check permission for FieldNumL
        PermOKFieldNumL ← CheckFieldNumLPerm(M1,FieldNumL,KeyRef)
        If(PermOKFieldNumL ≠ 1)

            ResultFlag ← FieldNumLWritePermInvalid
 30         Output ResultFlag

            Return
        EndIf
        -----------------------------------------------------------------


 35     #Find the type attribute for FieldNumE
        TypeFieldNumE ← FindFieldNumType(M1OfExternal,FieldNumE)
```

```
       #Find the type attribute for FieldNumL
       TypeFieldNumL ← FindFieldNumType(M1,FieldNumL)

       # Check type attribute for both fields match
5      If(TypeFieldNumE ≠TypeFieldNumL)
           ResultFlag ← TypeMismatch
           Output ResultFlag
           Return
       EndIf
10

       -----------------------------------------------------------------
   -----------------------------------------------------------------
   Do this if the Refill Device is tranferring Count-remaining for Printer
   upgrades
15     # If the Type is count remaining, check that upgrade values
       associated with
       # the count remaining are valid. Refer to Section 28. for further
       details on
       # count remaining and upgrade value.
20     If(TypeFieldNumL = TYPE_COUNT_REMAINING)∧(TypeFieldNumE
       =TYPE_COUNT_REMAINING)
           #Upgrade value field is lower adjoining field
           UpgradeValueFieldNumE = FieldNumE -1
           If(UpgradeValueFieldNumE < 0) # upgrade field doesn't exist for
25     QA Device being upgraded
               ResultFlag ← UpgradeFieldEInvalid
               Output ResultFlag
               Return
           EndIf
30         UpgradeValueFieldNumL = FieldNumL - 1
           If(UpgradeValueFieldNumL < 0) # upgrade field doesn't exist for
           local device
               ResultFlag ← UpgradeFieldLInvalid
               Output ResultFlag
35             Return
           EndIf
```

```
            UpgradeValueCheckOK ←
         UpgradeValCheck(UpgradeValueFieldNumL,M0,M1,

            UpgradeValueFieldNumL,M0OfExternal,M1OfExternal,KeyRef)
 5          If(UpgradeValueCheckOK = 0)
                ResultFlag ← UpgradeFieldMismatch
                Output ResultFlag
                Return
            EndIf
10       EndIf
         # Do this if Field Type is Count Remaining........end
         ------------------------------------------------------------------


15       #Check whether the device being upgraded can hold the transfer
         amount
         #(XferVal + AmountLeft
         OverFlow ← CanHold(FieldNumE,M0OfExternal,XferVal)
         If OverFlow error
20          ResultFlag ← FieldNumESizeInsufficient
            Output ResultFlag
            Return
         EndIf
         ------------------------------------------------------------------
25

         #Check the refill device has the desired amount (XferVal < =
         AmountLeft)
         UnderFlow ← HasAmount(FieldNumL,M0,XferVal)
         If UnderFlow error
30          ResultFlag ← FieldNumLAmountInsufficient
            Output ResultFlag
            Return
         EndIf
         ------------------------------------------------------------------
35       # All checks complete .....

         # Generate Seqdata for SEQ_1 and SEQ_2 fields
```

```
XferSEQ_1DataToDevice = XferSEQ_1DataFromDevice - 2
XferSEQ_2DataToDevice = XferSEQ_2DataFromDevice - 1


# Add DataSet to Xfer Entry Cache
AddDataSetToXferEntryCache(ChipId,FieldNumE, FieldNumL,
XferLength, XferVal, XferSEQ_1DataFromDevice,
XferSEQ_2DataFromDevice)


# Get current FieldDataE field data words to write to Xfer Entry
cache
GetFieldDataWords(FieldNumE,FieldDataE,M0OfExternal,M1OfExternal)


#Deduct XferVal from FieldNumL and Write new value
DeductAndWriteValToFieldNumL(XferVal,FieldNumL,M0)


#Generate new field data words for FieldNumE. The current
FieldDataE is added to
# XferVal to generate new FieldDataE
GenerateNewFieldData(FieldNumE,XferVal,FieldDataE)


# Generate FieldSelect and FieldVal for SeqData field SEQ_1, SEQ_2
and
# FieldDataE...
CurrentFieldSelect← 0
FieldVal ← 0
GenerateFieldSelectAndFieldVal(FieldNumE, FieldDataE,
XferSEQ_1FieldNum, XferSEQ_1DataToDevice,XferSEQ_2FieldNum,
XferSEQ_2DataToDevice,
FieldSelect,FieldVal)


#Generate message for passing into GenerateSignature function
data ← (RWSense|FieldSelect|ChipId|FieldVal)# Refer to Figure 373.
#Create output signature for FieldNumE
SIG_{out}← GenerateSignature(KeyRef,data,R_{L2},R_{E2})
```

```
        Update R_L2 to R_L3
        ResultFlag ← Pass
        Output ResultFlag, FieldData, R_L2 ,SIG_out
        Return
5       EndIf
```

27.1.4.1 *ResultFlag ValidateKeyRefAndSignature(KeyRef,data,R_E,R_L)*

This function checks KeyRef is valid, and if KeyRef is valid, then input signature is verified using KeyRef.

```
        CheckRange(KeyRef.keyNum)
10      If invalid
            ResultFlag ← InValidKey
            Output ResultFlag
            Return
        EndIf
15


        #Generate message for passing into GenerateSignature function
        data ← (RWSense|MSelect|KeyIdSelect|ChipId|WordSelect|M0|M1)
                # Refer to Figure 382.
20      #Generate Signature
        SIG_L ← GenerateSignature(KeyRef,data,R_E,R_L)


        # Check input signature SIG_E
        If(SIG_L = SIG_E)
25          Update R_L to R_L2
        Else
            ResultFlag ← Bad Signature
            Output ResultFlag
            Return
30      EndIf
```

27.1.4.2 *GenerateFieldSelectAndFieldVal (FieldNumE, FieldDataE, XferSEQ_1FieldNum, XferSEQ_1DataToDevice, XferSEQ_2FieldNum, XferSEQ_2DataToDevice,   FieldSelect, FieldVal)*

This functions generates the FieldSelect and FieldVal for output from FieldNumE and its final data,
35 and data to be written to Seq fields SEQ_1 and SEQ_2.

27.1.4.3 *PresentFlag GetFieldPresent(M1,FieldNum)*

This function checks whether FieldNum is a valid.

```
FieldSize[16] ← 0 # Array to hold FieldSize assuming there are 16
fields

NumFields← FindNumberOfFieldsInM0(M1,FieldSize) #Refer to Section
19.4.1
If(FieldNum< NumFields)
    PresentFlag← 1
Else
    PresentFlag← 0
EndIf
Return PresentFlag
```

27.1.4.4 *NumFields FindNumOfFieldsInM0(*M1*,FieldSize[])*

Refer to Figure 19.4.1 for details.

27.1.4.5 *FieldNum GetFieldNum(M1, Type)*

This function returns the field number based on the Type.

```
FieldSize[16] ← 0 # Array to hold FieldSize assuming there are 16
fields
NumFields← FindNumberOfFieldsInM0(M1,FieldSize) #Refer to Section
19.4.1
For i = 0 to NumFields
    If(M1[i].Type = Type)
        Return i # This is field Num for matching field
EndFor
i = 255 # If XferSession field was not found then return an
invalid value
Return i
```

27.1.4.6 *PermOK CheckFieldNumEPerm(M1,FieldNumE)*

This function checks authenticated write permission for FieldNum which holds the upgraded value.

```
AuthRW ← M1[FieldNum].AuthRW
NonAuthRW ← M1[FieldNum].NonAuthRW
If(AuthRW = 1) ∧(NonAuthRW = 0)
    PermOK ← 1
Else
    PermOK ← 0
EndIf
Return PermOK
```

27.1.4.7 PermOK CheckSeqDataFieldPerms(M1, XferSEQ_1FieldNum, XferSEQ_2FieldNum, FieldNumE)

This function checks that both SeqData fields have Decrement-Only permission with the same key that has write permission on FieldNumE.

```
5        KeyNumForFieldNumE ← M1[FieldNumE].KeyNum # Isolate KeyNum for the
         field that will

                                                              # be upgraded
         # Isolate KeyNum for both SeqData fields and check that they can
         be written using the same key
10       KeyNumForSEQ_1 ← M1[XferSEQ_1FieldNum].KeyNum
         KeyNumForSEQ_2 ← M1[XferSEQ_2FieldNum].KeyNum
         If(KeyNumForSEQ_1 ≠ KeyNumForSEQ_2)
             PermOK ← 0
             Return PermOK
15       EndIf
         # Check that the write key for FieldNumE and SeqData field is not
         the same
         If (KeyNumForSEQ_1 = KeyNumForFieldNumE)
             PermOK ← 0
20           Return PermOK
         EndIf
         #Isolate Decrement-Only permissions with the write key of
         FieldNumE
         KeyPermsSEQ_1 ← M1[XferSEQ_1FieldNum].KeyPerms[KeyNumForFieldNumE]
25       KeyPermsSEQ_2 ← M1[XferSEQ_2FieldNum].KeyPerms[KeyNumForFieldNumE]
         # Check that both sequence fields have Decrement-Only permission
         for this key
         If(KeyPermsSEQ_1 = 0) ∨ (KeyPermsSEQ_2 = 0)
             PermOK ← 0
30           Return PermOK
         EndIf
         PermOK ← 1
         Return PermOK
```

27.1.4.8 AddDataSetToXferEntryCache (ChipId, FieldNumE, FieldNumL,
35   XferVal, SEQ_1Data, SEQ_2Data)

This function adds a new dataset to the Xfer Entry cache. Dataset is a single record in the Xfer Entrycache. Refer to Section 27 for details.

```
# Search for matching ChipId FieldNumE is Cache
DataSet ←SearchDataSetInCache (ChipId, FieldNumE)
# If found
If(DataSet is valid)
    DeleteDataSetInCache(DataSet) # This creates a vacant dataset
    AddRecordToCache(ChipId, FieldNumE,FieldDataL,XferVal,SEQ_1Data,
SEQ_2Data)
EndIf
# Searches the cache for XferState complete/deleted
Found← SearchRecordsInCache(complete/deleted)
If(Found =1)
    AddRecordToCache(ChipId, FieldNumE,FieldDataL,XferVal,SEQ_1Data,
SEQ_2Data)
Else
    # This will overwrite the oldest DataSet in cache
    AddRecordToCache(ChipId, FieldNumE,FieldDataL,XferVal,SEQ_1Data,
SEQ_2Data)
    Return
Endif
Set XferState in record to Xfer
Return
```

27.1.4.9 *FieldType FindFieldNumType(M1,FieldNum)*

This function gets the Type attribute for a given field.

```
FieldType ← M1[FieldNum].Type
Return FieldType
```

27.1.4.10 *PermOK CheckFieldNumLPerm(M1,FieldNumL,KeyRef)*

This function checks authenticated write permissions using KeyRef for FieldNumL in the refill device.

```
AuthRW ← M1[FieldNumL].AuthRW
KeyNumAtt ← M1[FieldNumL].KeyNum
DOForKeys ← M1[FieldNumL].DOForKeys[KeyNum]
# Authenticated write allowed
# ReadWrite key for field is the same as Input KeyRef.keyNum
# Key has both ReadWrite and DecrementOnly Permission
```

```
If(AuthRW = 1) ∧ (KeyRef.keyNum = KeyNumAtt) ∧ (DOForKeys = 1

    PermOK← 1

Else

    PermOK← 0
```

5
```
EndIf

Return PermOK
```

27.1.4.11 CheckOK UpgradeValCheck(FieldNum1, M0OfFieldNum1, M1OfFieldNum1, FieldNum2, M0OfFieldNum2, M1OfFieldNum2,KeyRef)

This function checks the upgrade value corresponding to the count remaining. The upgrade value

10 corresponding to the count remaining field is stored in the lower adjoining field. To upgrade the

count remaining field, the upgrade value in refill device and the device being upgraded must match.

```
    #Check authenticated write permissions is allowed to the field

    #Check that only one key has ReadWrite access,

    #and all other keys are ReadOnly access
```

15
```
    PermCheckOKFieldNum1

    ←CheckUpgradeKeyForField(FieldNum1,M1OfFieldNum1,KeyRef)

    If(PermCheckOKFieldNum1 ≠ 1)

        CheckOK ← 0

        Return CheckOK
```

20
```
    EndIf


    PermCheckOKFieldNum2

    ←CheckUpgradeKeyForField(FieldNum2,M1OfFieldNum2,KeyRef)
```

25
```
    If(PermCheckOKFieldNum2 ≠ 1)

        CheckOK ← 0

        Return CheckOK

    EndIf
```

30
```
    #Get the upgrade value associated with field

    GetFieldDataWords(FieldNum1,UpgradeValueFieldNum1,M0OfFieldNum1,M1
    OfFieldNum1)


    #Get the upgrade value associated with field
```
35
```
    GetFieldDataWords(FieldNum2,UpgradeValueFieldNum2,M0OfFieldNum2,M1
    OfFieldNum2)
```

```
If(UpgradeValueFieldNum1 ≠ UpgradeValueFieldNum2)

    CheckOK ← 0

    Return CheckOK

EndIf
```

Get the type attribute for the field

```
UpgradeTypeFieldNum1← GetUpgradeType(FieldNum1,M1OfFieldNum1)

UpgradeTypeFieldNum2← GetUpgradeType(FieldNum2,M1OfFieldNum2)

If(UpgradeTypeFieldNum1 ≠ UpgradeTypeFieldNum2)

    CheckOK ← 0

    Return CheckOK

EndIf

CheckOK ← 1

Return CheckOK
```

27.1.4.12 CheckOK CheckUpgradeKeyForField(FieldNum,M1,KeyRef)

This function checks that authenticated write permissions is allowed to the field. It also checks that only one key has ReadWrite access and all other keys have ReadOnly access. KeyRef which updates count remaining must not have write access to the upgarde value field.

```
KeyNum ←M1[FieldNum].KeyNum

AuthRW ← M1[FieldNum].AuthRW

NonAuthRW ← M1[FieldNum].NonAuthRW

DOForKeys← M1[FieldNum].DOForKeys
```

#Check that KeyRef doesn't have write permissions to the field

```
If(KeyRef.keyNum = KeyNum)

    CheckOK ← 0

    Return CheckOK

EndIf
```

#AuthRW access allowed or NonAuthRW not allowed

```
If(AuthRW = 0) ∨ (NonAuthRW =1)

    CheckOK ← 0

    Return CheckOK

EndIf

For i ← 0 to 7
```

    # Keys other than KeyNum are allowed ReadOnly access,
    # DecrementOnly access not allowed for other keys(not KeyNum)

```
    If (i ≠ KeyNum) ∧(DOForKeys[i] = 1)

        CheckOK ← 0
```

```
                    Return CheckOK
                EndIf
                #ReadWrite access allowed for KeyNum,
                #ReadWrite and DecrementOnly access not allowed for KeyNum.
5               If (i = KeyNum) ∧(DOForKeys[i] = 1)
                    CheckOK ← 0
                    Return CheckOK
                EndIf
            EndFor
10          CheckOK ← 1
            Return CheckOK
```

27.1.4.13 UpgradeType GetUpgradeType(FieldNum, M1)

This function gets the type attribute for the upgrade field.

```
            UpgradeType GetUpgradeType(FieldNum)
15          UpgradeType←M1[FieldNum].Type
            Return UpgradeType
```

27.1.4.14 GetFieldDataWords(FieldNum,FieldData[], M0,M1)

This function gets the words corresponding to a given field.

```
            CurrPos ← MaxWordInM
20          If FieldNum = 0
                CurrPos ← MaxWordInM
            Else
                CurrPos ← (M1[FieldNum -1].EndPos) -1 # Next lower word after
            last word of the
25                                                      # previous
            field
            EndIf
            EndPos ← (M1[FieldNum].EndPos)
            For i ← EndPos to CurrPos j ← 0
30              FieldData[j] ← M0[i]  #Copy M0 word to FieldData array
            EndFor
```

27.2   STARTROLLBACK

Input: *KeyRef, $_{M0}$OfExternal, $_{M1}$OfExternal, ChipId, FieldNumL,*

FieldNumE, InputParameterCheck (optional), R_E, SIG_E, R_{E2}

35   *Output:*   *ResultFlag, FieldSelect, FieldVal, R_{L2}, SIG_{out}*

Changes: *$_{M0}$ and R_L*

27.2.1   Function description

StartRollBack function is used to start a *rollback sequence* if the QA Device being upgraded didn't receive the transfer message correctly and hence didn't receive the transfer.

5   The system calls the function on the upgrading QA Device, passing in *FieldNumE* and *ChipId* of the QA Device being upgraded, and *FieldNumL* of the upgrading QA Device. The upgrading QA Device checks that the QA Device being upgraded didn't actually receive the message correctly, by comparing the values read from the device with the values stored in the *Xfer Entry* cache. The values compared is the value of the *sequence* fields. After all checks are fulfilled, the upgrading QA

10   Device produces the new data for the sequence fields and a signature. This is subsequently applied to the QA Device being upgraded (using the *WriteFieldAuth* function), which updates the *sequence fields SEQ_1 and SEQ_2* to the pre-rollback values. However, the new data for the sequence fields and signature can only be applied if the previous data for the sequence fields produced by *Xfer* function has not been written.

15   The output from the StartRollBack function consists only of the field data of the two sequence fields, and a signature using the refill key. When a pre-rollback output is produced, then sequence field data in *SEQ_1 (as stored in the Xfer Entry cache, which is what is passed in to the XferAmount function)* is decremented by 1 and the sequence field data in *SEQ_2 (as stored in the Xfer Entry cache, which is what is passed in to the XferAmount function)* is decremented by 2.

20   Additional *InputParameterCheck* value must be provided for the parameters not included in the SIG_E, if the transmission between the System and Ink Refill QA Device is error prone, and these errors are not corrected by the transimission protocol itself. *InputParameterCheck* is *SHA-1[FieldNumL | FieldNumE]*, and is required to ensure the integrity of these parameters, when these inputs are received by the Ink Refill QA Device.

25   The *StartRollBack* function must first calculate the *SHA-1[FieldNumL | FieldNumE]*, compare the calculated value to the value received (*InputParameterCheck*) and only if the values match act upon the inputs.

27.2.2   Input parameters

30   Table 293 describes each of the input parameters for StartRollback function.

Parameter	Description
KeyRef	For common key input signature: KeyRef.keyNum = Slot number of the key to be used for testing input signature. SIG_E produced using $K_{KeyRef.keyNum}$ by the QA Device being upgraded. KeyRef.useChipId = 0
	For variant key input signature: KeyRef.keyNum = Slot number of the key to be

	used for generating the variant key for testing input signature. SIG_E produced using a variant of $K_{KeyRef.keyNum}$ by the QA Device being upgraded. $KeyRef.useChipId = 1$. $KeyRef.chipId$ = ChipId of the device which generated SIG_E.
$M_0OfExternal$	All 16 words of M_0 of the QA Device being upgraded which failed to upgrade.
$M_1OfExternal$	All 16 words of M_1 of the QA Device being upgraded which failed to upgrade.
ChipId	ChipId of the QA Device being upgraded which failed to upgrade.
FieldNumL	M_0 field number of the local (refill) device from which the value was supposed to transferred.
FieldNumE	M_0 field number of the QA Device being upgraded to which the value couldn't be transferred.
R_E	External random value used to verify input signature. This will be the R from the input signature generator (i.e device generating SIG_E). The input signal generator in this case, is the device which failed to upgrade or a translation device.
SIG_E	External signature required for authenticating input data. The input data in this case, is the output from the Read function performed on the device which failed to upgrade. A correct $SIG_E = SIG_{KeyRef}(Data \mid R_E \mid R_L)$.

27.2.2.1 Input signature verification data format

Refer to Section 27.1.2.1.

27.2.3 Output parameters

Table 294 describes each of the output parameters for StartRollback function.

Parameter	Description
ResultFlag	Indicates whether the function completed successfully or not. If it did not complete successfully, the reason for the failure is returned here. See Section 12.1,Table 292 and Table 295.
FieldSelect	Selection of fields to be written. In this case the bits corresponding to SEQ_1 and SEQ_2 are set to 1. All other bits are set to 0.
FieldVal	Updated data for sequence datat field for QA Device being upgraded. This must be passed as input to the WriteFieldsAuth function of the QA Device being upgraded.
R_{L2}	Internal random value required to generate output signature. This must be passed as input to the WriteFieldsAuth function or Translate

	function of the QA Device being upgraded.
SIG_{out}	Output signature which must be passed as an input to the *WriteFieldsAuth* function of the QA Device being upgraded. $SIG_{out} = SIG_{KeyRef}(\text{data} \mid R_{L2} \mid R_{E2})$ as per Figure 373.

Table 295. Result definition for StartRollBack

ResultFlag Definition	Description
RollBackInvalid	RollBack cannot be performed on the request because parameters for rollback is incorrect.

27.2.3.1 SIG_{Out}

Refer to Section 20.2.1 for details.

27.2.4 Function sequence

The *StartRollBack* command is illustrated by the following pseudocode:

Accept input parameters-KeyRef, M0OfExternal, M1OfExternal, ChipId, FieldNumL, FieldNumE, R_E, SIG_E, R_{E2}

```
Accept R_E, SIG_E, R_E2
#Generate message for passing into ValidateKeyRefAndSignature
function
data ← (RWSense|MSelect|KeyIdSelect|ChipId|WordSelect|M0|M1)
        # Refer to Figure 382.

-------------------------------------------------------------
# Validate KeyRef, and then verify signature
ResultFlag = ValidateKeyRefAndSignature(KeyRef,data,R_E,R_L)
If (ResultFlag ≠ Pass)
   Output ResultFlag
   Return
EndIf
-----------------------------------------------------------#
Check Seq Fields Exist and get their Field Num
# Get Seqdata field SEQ_1 num for the device being upgraded
XferSEQ_1FieldNum← GetFieldNum(M1OfExternal, SEQ_1)
```

```
        # Check if the Seqdata field SEQ_1 is valid
        If(XferSEQ_1FieldNum invalid)
            ResultFlag ← SeqFieldInvalid
5           Output ResultFlag
            Return
        EndIf
        # Get Seqdata field SEQ_2 num for the device being upgraded
        XferSEQ_2FieldNum← GetFieldNum(M1OfExternal, SEQ_2)
10

        # Check if the Seqdata field SEQ_2 is valid
        If(XferSEQ_2FieldNum invalid)
            ResultFlag ← SeqFieldInvalid
            Output ResultFlag
15          Return
        EndIf


        ------------------------------------------------------------------
        # Get SeqData SEQ_1 data from device being upgraded
20      GetFieldDataWords(XferSEQ_1FieldNum,

            XferSEQ_1DataFromDevice,M0OfExternal,M1OfExternal)

        # Get SeqData SEQ_2 data from device being upgraded
25      GetFieldDataWords(XferSEQ_2FieldNum,
                          XferSEQ_2DataFromDevice,
        M0OfExternal,M1OfExternal)


        ------------------------------------------------------------------
30      # Check Xfer Entry in cache is correct - dataset exists, Field
        data
        # and sequence field data matches and Xfer State is correct
        XferEntryOK ← CheckEntry(ChipId, FieldNumE, FieldNumL,
                          XferSEQ_1DataFromDevice, XferSEQ_2DataFromDevice)
35

        If( XferEntryOK= 0)
            ResultFlag ← RollBackInvalid
```

850

```
        Output ResultFlag
        Return
    EndIf
```

5

```
    # Generate Seqdata for SEQ_1 and SEQ_2 fields
    XferSEQ_1DataToDevice = XferSEQ_1DataFromDevice - 1
    XferSEQ_2DataToDevice = XferSEQ_2DataFromDevice - 2
```

10

```
    # Generate FieldSelect and FieldVal for sequence fields SEQ_1 and
    SEQ_2
    CurrentFieldSelect← 0
    FieldVal ← 0
    GenerateFieldSelectAndFieldVal(XferSEQ_1FieldNum,
```

15

```
    XferSEQ_1DataToDevice, XferSEQ_2FieldNum, XferSEQ_2DataToDevice,
    FieldSelect, FieldVal)
```

```
    #Generate message for passing into GenerateSignature function
    data ← (RWSense|FieldSelect|ChipId|FieldVal)# Refer to Figure 373.
```

20

```
    #Create output signature for FieldNumE
    SIG_{out}← GenerateSignature(KeyRef,data,R_{L2},R_{E2})
    Update R_{L2} to R_{L3}
    ResultFlag ← Pass
    Output ResultFlag, FieldData, R_{L2} ,SIG_{out}
```

25

```
    Return
    EndIf
```

27.3   ROLLBACKAMOUNT

	Input:	KeyRef, $_{M0}$OfExternal, $_{M1}$OfExternal, ChipId, FieldNumL, FieldNumE, InputParameterCheck (optional), R_E, SIG_E
30	Output:	ResultFlag
	Changes:	$_{M0}$ and R_L
	Availablity:	Ink refill QA Device

27.3.1   Function description

RollBackAmount function finally adjusts the value of the *FieldNumL* of the upgarding QA Device to a

35   previous value before the transfer request, if the QA Device being upgraded didn't receive the

transfer message correctly (and hence was not upgraded).

The upgrading QA Device checks that the QA Device being upgraded didn't actually receive the transfer message correctly, by comparing the sequence data field values read from the device with the values stored in the *Xfer Entry* cache. The sequence data field values read must match what was previously written using the *StartRollBack* function. After all checks are fulfilled, the upgrading

5   QA Device adjusts its *FieldNumL*.

Additional *InputParameterCheck* value must be provided for the parameters not included in the SIG_E, if the transmission between the System and Ink Refill QA Device is error prone, and these errors are not corrected by the transimission protocol itself. *InputParameterCheck* is *SHA-1[FieldNumL | FieldNumE]*, and is required to ensure the integrity of these parameters, when these

10   inputs are received by the Ink Refill QA Device.

The *RollBackAmount* function must first calculate the *SHA-1[FieldNumL | FieldNumE]*, compare the calculated value to the value received (*InputParameterCheck*) and only if the values match act upon the inputs.

27.3.2   Input parameters

15   Table 296 describes each of the input parameters for RollbackAmount function.

Parameter	Description
KeyRef	*For common key input signature: KeyRef.keyNum* = Slot number of the key to be used for testing input signature. SIG_E produced using $K_{KeyRef.keyNum}$ by the QA Device being upgraded. *KeyRef.useChipId* = 0
	For variant key input signature: KeyRef.keyNum = Slot number of the key to be used for generating the variant key for testing input signature. SIG_E produced using a variant of $K_{KeyRef.keyNum}$ by the QA Device being upgraded. *KeyRef.useChipId* = 1 *KeyRef.chipId* = ChipId of the device which generated SIG_E.
$_{M0}$*OfExternal*	All 16 words of $_{M0}$ of the QA Device being upgraded which failed to upgrade.
$_{M1}$*OfExternal*	All 16 words of $_{M1}$ of the QA Device being upgraded which failed to upgrade.
ChipId	ChipId of the QA Device being upgraded which failed to upgrade.
FieldNumL	$_{M0}$ field number of the local (refill) device from which the value was supposed to transferred.
FieldNumE	$_{M0}$ field number of the QA Device being upgraded to which the value was not transferred.
R_E	External random value used to verify input signature. This will be the R from the input signature generator (i.e device generating SIG_E). The input signal generator in this case, is the device which failed to upgrade or a translation device.
SIG_E	External signature required for authenticating input data. The input data in this

	case, is the output from the *Read* function performed on the device which failed to upgrade. A correct SIG_E = SIG$_{KeyRef}$(Data | R$_E$ | R$_L$).

27.3.2.1 Input signature generation data format

Refer to Section 27.1.2.1 for details.

27.3.3 Output parameters

Table 297 describes each of the output parameters for RollbackAmount.

Parameter	Description
ResultFlag	Indicates whether the function completed successfully or not. If it did not complete successfully, the reason for the failure is returned here. See Section 12.1, Table 292 and Table 295.

27.3.4 Function sequence

The *RollBackAmount* command is illustrated by the following pseudocode:

```
Accept input parameters-KeyRef, M0OfExternal, M1OfExternal,
ChipId, FieldNumL, FieldNumE, R_E,SIG_E


#Generate message for passing into ValidateKeyRefAndSignature
function
data ← (RWSense|MSelect|KeyIdSelect|ChipId|WordSelect|M0|M1)
        # Refer to Figure 382.
-------------------------------------------------------------------
# Validate KeyRef, and then verify signature
ResultFlag = ValidateKeyRefAndSignature(KeyRef,data,R_E,R_L)
If (ResultFlag ≠ Pass)
   Output ResultFlag
   Return
EndIf
-------------------------------------------------------------------
# Check Seq Fields Exist and get their Field Num
# Get Seqdata field SEQ_1 num for the device being upgraded
XferSEQ_1FieldNum← GetFieldNum(M1OfExternal, SEQ_1)


# Check if the Seqdata field SEQ_1 is valid
```

853

```
        If(XferSEQ_1FieldNum invalid)
           ResultFlag  ← SeqFieldInvalid
           Output ResultFlag
           Return
5       EndIf
        # Get Seqdata field SEQ_2 num for the device being upgraded
        XferSEQ_2FieldNum← GetFieldNum(M1OfExternal, SEQ_2)


        # Check if the Seqdata field SEQ_2 is valid
10      If(XferSEQ_2FieldNum invalid)
           ResultFlag  ← SeqFieldInvalid
           Output ResultFlag
           Return
        EndIf
15

        ------------------------------------------------------------------
        # Get SeqData SEQ_1 data from device being upgraded
        GetFieldDataWords(XferSEQ_1FieldNum,

20          XferSEQ_1DataFromDevice,M0OfExternal,M1OfExternal)

        # Get SeqData SEQ_2 data from device being upgraded
        GetFieldDataWords(XferSEQ_2FieldNum,
                          XferSEQ_2DataFromDevice,
25      M0OfExternal,M1OfExternal)

        -----------------------------------------------------------------
        # Generate Seqdata for SEQ_1 and SEQ_2 fields with the data that
        is read
30      XferSEQ_1Data = XferSEQ_1DataFromDevice + 1
        XferSEQ_2Data = XferSEQ_2DataFromDevice + 2

        # Check Xfer Entry in cache is correct - dataset exists, Field
        data
35      # and sequence field data matches and Xfer State is correct
        XferEntryOK  ← CheckEntry(ChipId, FieldNumE, FieldNumL,
                          XferSEQ_1Data, XferSEQ_2Data)
```

```
      If( XferEntryOK= 0)
         ResultFlag ← RollBackInvalid
         Output ResultFlag
5        Return
      EndIf
      # Get ΔFieldDataL from DataSet
      GetVal(ChipId, FieldNumE,ΔFieldDataL)
      # Add ΔFieldDataL to FieldNumL
10    AddValToField(FieldNumL,ΔFieldDataL)
      # Update XferState in DataSet to complete/deleted
      UpdateXferStateToComplete(ChipId,FieldNumE)
      ResultFlag ← Pass
      Output ResultFlag
15    Return
```

FUNCTIONS

UPGRADE DEVICE

(PRINTER UPGRADE)

28   Concepts

5   This section is very similar to Section 26. The differences between this section and Section 26 have been summarised and underlined, where required.

28.1   PURPOSE

In a printing application, a printer contains a Printer QA Device, which stores details of the various operating parameters of a printer, some of which may be upgradeable. The upgradeable

10   parameters must be written (initially) and changed in an authorised manner.

The authorisation for the write or change is achieved by using a *Parameter Upgrader QA Device* which contains the necessary functions to allow a write or a change of a parameter value (e.g. a print speed) into another QA Device, typically a printer QA Device. This QA Device is also referred to as an upgrading QA Device.

15   A parameter upgrader QA Device is able to perform a fixed number of upgrades, and this number is effectively a consumable value. The number of upgrades remaining is also referred to *as count-remaining*. With each write/change of an operating parameter in a *Printer QA Device,* the count-remaining decreases by 1, and can be replenished by a value upgrader QA Device.

The *Parameter Upgrader QA Device* can also be referred to as the *Upgrading QA Device*, and the

20   *Printer QA Device* can also be referred to as the *QA Device being upgraded.*

The writing or changing of the parameter can also be referred to as a transfer of a parameter.

The Parameter Upgrader QA Device copies its parameter value field to the parameter value field of Printer QA Device, and decrements the count-remaining field associated with the parameter value field by 1.

25   28.2   REQUIREMENTS

The transfer of a parameter has two basic requirements:

- The transfer can only be performed if the transfer request is valid. The validity of the transfer request must be completely checked by the Parameter Upgrader QA Device, before it produces the required output for the transfer. It must not be possible to apply the transfer

30   output to the Printer QA Device, if the Parameter Upgrader QA Device has been already been rolled back for that particular transfer.

- A process of rollback is available if the transfer was not received by the Printer QA Device. A rollback is performed only if the rollback request is valid. The validity of the rollback request must be completely checked by the Parameter Upgrader QA Device , before the

35   count-remaining value is incremented by 1. It must not be possible to rollback an Parameter Upgrader QA Device for a transfer, which has already been applied to the Printer QA

Device i.e the Parameter Upgrader QA Device must only be rolled back for transfers that have actually failed.

28.3   BASIC SCHEME

5   The transfer and rollback process is shown in Figure 383.

Following is a sequential description of the transfer and rollback process:

1.   The System *Reads* the memory vectors M0 and M1 of the Printer QA Device. The output from the read which includes the M0 and M1 words of the Printer QA Device, and a signature, is passed as an input to the *Transfer Request*. It is essential that M0 and M1are

10   read together. This ensures that the field information for M0 fields are correct, and have not been modified, or substituted from another device. Entire M0 and M1 must be read to verify the correctness of the subsequent *Transfer Request* by the Parameter Upgrader QA Device.

2.   The System makes a *Transfer Request* to the Parameter Upgrader QA Device with the field in the Parameter Upgrader QA Device whose data will be copied to the Printer QA Device,

15   and the field in Printer QA Device to which this data will be copied to. The *Transfer Request* also includes the output from Read of the Printer QA Device. The Parameter Upgrader QA Device validates the *Transfer Request* based on the *Read* output, checks that it has enough count-remaining for a successful transfer, and then produces the necessary *Transfer output*. The *Transfer Output* typically consists of new field data for the field being refilled or

20   upgraded, additional field data required to ensure the correctness of transfer/rollback, along with a signature.

3.   The System then applies the *Transfer Output* on the Printer QA Device, by calling an authenticated *Write* on it, passing in the *Transfer Output*. The Write is either successful or not. If the *Write* is not successful, then the System will repeat calling the *Write* function using

25   the same transfer output, which may be successful or not. If unsuccessful the System will initiate a *rollback* of the transfer. The *rollback* must be performed on the Parameter Upgrader QA Device, so that it can adjust its value to a previous value before the current *Transfer Request* was initiated.

4.   The System starts a *rollback by Reading* the memory vectors M0 and M1 of the Printer QA

30   Device.

5.   The System makes a *StartRollBack* Request to the Parameter Upgrader QA Device with same input parameters as the *Transfer Request*, and the output from *Read* in (4). The Parameter Upgrader QA Device validates the *StartRollBack* Request based on the *Read* output, and then produces the necessary *Pre-rollback* output. The *Pre-rollback* output

35   typically consists only of additional field data along with a signature.

6.   The System then applies the *Pre-rollback* output on the Parameter Upgrader QA Device, by calling an authenticated *Write* on it, passing in the *Pre-rollback* output. The Write is either

857

successful or not. If the Write is not successful, then either (6), or (5) and (6) must be repeated.

7. The System then *Reads* the memory vectors M0 and M1 of the Printer QA Device.

8. The System makes a *RollBack Request* to the Parameter Upgrader QA Device with same input parameters as the *Transfer Request*, and the output from *Read* (7). The Parameter Upgrader QA Device validates the *RollBack Request* based on the *Read* output, and then rolls back its count-remaining field by incrementing it by 1.

28.3.1 Transfer

The Printer QA Device stores upgradeable operating parameter values in M0 fields, and its corresponding M_1 words contains field information for its operating parameter fields. The field information consists of the size of the field, the Type of data stored in field and the access permission to the field. See Section 8.1.1 for details.

The *Parameter Upgrader QA Device* also stores the new operating parameter values (which will be written to the Printer QA Device) in its M0 fields, and its coressponding M_1 words contains field information for the new operating parameter fields. Additionally, *the Parameter Upgrader QA Device has a count-remaining field associated with the new operating parameter value field. The count-remaining field occupies the higher field position when compared to its associated operating parameter value field.*

28.3.1.1 Authorisation

The basic authorisation for a transfer comes from a *key*, which has authenticated ReadWrite permission (stored in field information as KeyNum) to the operating parameter field in the *Printer QA Device*. We will refer to this key as the *upgrade key*. The same *upgrade key* must also have authenticated decrement-only permission to the count-remaining field (which decrements by 1 with every transfer) in the *Parameter Upgrader QA Device*.

After validating the input upgrade request, the *Parameter Upgrader QA Device* will decrement the value of the count-remaining field by 1, and produce data (by copying the data stored from its operating parameter field) and signature for the new operating parameter using the *upgrade key*. *Note that the Parameter Upgrader QA Device can decrement its count-remaining field only if the upgrade key has the permission to decrement it.*

The data and signature produced by the *Parameter Upgrader* QA Device is subsequently applied to the *Printer QA Device*. The *Printer QA Device* will accept the new transferred operating parameter, only if the signature is valid. *Note that the signature will only be valid if it was produced using the upgrade key which has write permission to the operating parameter field being written.*

The upgrade key has authenticated ReadWrite permission to the operating parameter field (which will change) in the Printer QA Device. The upgrade key has decrement-only permission to the the count-remaining field (which decrements by 1 with every transfer of field) in the Parameter Upgrader QA Device.

858

28.3.1.2 Data Type matching

The *Parameter Upgrader* QA Device validates the transfer request by matching the *Type* of the data in the field information of operating parameter field (stored in M1) of Printer QA Device to the *Type* of data in the field information of operating parameter field of the *Parameter Upgrader* QA Device. This ensures that equivalent data types are being transferred i.e Network_OEM1_printspeed_1500 is not transferred to Network_OEM1_printspeed_2000.

28.3.1.3 Addition validation

Additional validation of the transfer request must be performed before a transfer output is generated by the *Parameter Upgrader* QA Device. These are as follows:

- For the Printer QA Device

1. Whether the field being upgraded is actually present.
2. Whether the field being upgraded can hold the changed value.

- For the *Parameter Upgrader* QA Device:

1. Whether the new operating parameter field and its associated count-remaining is actually present.
2. Whether the count-remaining field has an upgrade left for the transfer to succeed.

28.3.1.4 Rollback facilitation

To facilitate a rollback, the Parameter Upgrade QA Device will store a *list of transfer requests* processed by it. This list is referred to as the *Xfer Entry* cache. Each record in the list consists of the transfer parameters corresponding to the transfer request.

28.3.2 Rollback

A rollback request will be validated by looking through the Xfer Entry cache of the Parameter Upgrader QA Device. After the right transfer request is found the Parameter Upgrade QA Device checks that the output from the transfer request was not applied to the Printer QA Device by comparing the current Read of the Printer QA Device to the values in the Xfer Entry cache, and finally rolling back the Parameter Upgrader QA Device count-remaining field by incrementing it by 1. The *Parameter Upgrader* QA Device must be absolutely sure that the Printer QA Device didn't receive the transfer. This factor determines the additional fields that must be written along with new operating parameter data, and also the parameters of the transfer request that must be stored in the Xfer Entry cache to facilitate a rollback, to prove that the Printer QA Device didn't actually receive the transfer.

The rollback process increments the count-remaining field by 1 in the Parameter Upgrader QA Device.

28.3.2.1 Sequence fields

The rollback process must ensure that the transfer output (which was previously produced) for which the rollback is being performed, cannot be applied after the rollback has been performed.

How do we achieve this? There are two separate decrement-only *sequence* fields (*SEQ_1* and *SEQ_2*) in the Printer QA Device which can only be decremented by the Parameter Upgrader QA Device using the *upgrade key*. The nature of data to be written to the sequence fields is such that either the transfer output or the pre-rollback output can be applied to the Printer QA Device, but not

5   both i.e they must be mutually exclusive. Refer to Table 285 for details.

The two sequence fields are initialised to 0xFFFFFFFF using *sequence key.*The sequence key is different to the upgrade key, and has authenticated ReadWrite permission to both the sequence fields.

The transfer output consists of the new data for the field being upgraded, field data of the two

10   sequence fields, and a signature using the upgrade key. The field data for SEQ_1 is decremented by 2 from the original value that was passed in with the transfer request. The field data for SEQ_2 is decremented by 1 from the original value that was passed in with the transfer request.

The pre-rollback output consists only of the field data for the two sequence fields, and a signature using the upgrade key. The field data for SEQ_1 is decremented by 1 from the original value that

15   was passed in with the transfer request. The field data for SEQ_2 is decremented by 2 from the original value that was passed in with the transfer request.

Since the two sequence fields are decrement-only fields, the writing of the transfer output to QA Device being upgraded will prevent the writing of the pre-rollback output to QA Device being upgraded, since the sequence fields are decrement-only fields, and only one possible set can be

20   written. If the writing of the transfer output fails, then pre-rollback can be written. However, the transfer output cannot be written after the pre-rollback output has been written.

Before a rollback is performed, the Parameter Upgrader QA Device must confirm that the sequence fields was successfully written to the pre-rollback values in the Printer QA Device. Because the sequence fields are decrement-only fields, the Printer QA Device will allow pre-rollback output to be

25   written only if the transfer output has not been written.

28.3.2.1.1   Field information of the sequence data field

For a device to be upgradeable the device must have two sequence fields SEQ_1 and SEQ_2 which are written with sequence data during the *transfer sequence*. Thus all upgrading QA Devices, ink QA Devices and printer QA Devices must have two *sequence* fields. The upgrading QA Devices

30   must have these fields because they can be upgraded as well. The *sequence* field information are defined in Table 298.

Attribute Name	Value	Explanation
Type	TYPE_SEQ_1 or TYPE_SEQ_2.	See Appendix A for exact data.
KeyNum	Slot number of the *sequence key.*	Only the *sequence key* has authenticated *ReadWrite* access to this field.
Non Auth RW Perm[b]	0	Non authenticated ReadWrite is not allowed to the field.
Auth RW Perm[c]	1	Authenticated (key based) ReadWrite access is allowed to the field.
KeyPerm	KeyPerms[KeyNum] = 0	KeyNum is the slot number of the *sequence* key, which has *ReadWrite* permission to the field.
	KeyPerms[Slot number of upgrade key] = 1	*Upgrade key* can *decrement* the sequence field.
	KeyPerms[others= 0 ..7(except upgrade key)] = 0	All other keys have *ReadOnly* access.
End Pos		Set as required. Size is typically 1 word.

a. This is a sample type only and is not included in the Type Map in Appendix A.

b. Non authenticated Read Write permission.

c. Authenticated Read Write permission.

28.3.3   Upgrade states

There are three states in an transfer sequence, the first state is initiated for every transfer, while the next two states are initiated only when the transfer fails. The states are - Xfer, StartRollback, and Rollback.

28.3.3.1   Upgrade Flow

Figure 384 shows a typical upgrade flow.

28.3.3.2   Xfer

This state indicates the start of the transfer process, and is the only state required if the transfer is successful. During this state, the Parameter Upgrader QA Device adds a new record to its Xfer Entry cache, decrements its count-remaining by 1, produces new operating parameter field, new sequence data (as described in Section 28.3.2.1) and a signature based on the upgrade key.

The Printer QA Device will subsequently write the new operating parameter field and new sequence data, after verifying the signature. If the new operating parameter field can be successfully written to the Printer QA Device, then this will finish a successful transfer.

If the writing of the new amount is unsuccessful (result returned is BAD SIG), the System will re-transmit the transfer output to the Printer QA Device, by calling the authenticated Write function on it again, using the same transfer output.

If retrying to write the same transfer output fails repeatedly, the System will start the rollback process on Parameter Upgrader QA Device, by calling the Read function on the Printer QA Device, and subsequently calling the StartRollBack function on the Parameter Upgrader QA Device. After a successful rollback is performed, the System will invoke the transfer sequence again.

28.3.3.3 StartRollBack

This state indicates the start of the rollback process. During this state, the Parameter Upgrade QA Device produces the next sequence data and a signature based on the *upgrade key*. This is also called a pre-rollback, as described in Section 26.3.2.

The pre-rollback output can only be written to the Printer QA Device, if the previous transfer output has not been written. The writing of the *pre-rollback* sequence data also ensures, that if the previous transfer output was captured and not applied, then it cannot be applied to the Printer QA Device in the future.

If the writing of the pre-rollback output is unsuccessful (result returned is BAD SIG), the System will re-transmit the pre-rollback output to the Printer QA Device, by calling the authenticated Write function on it again, using the same pre-rollback output.

If retrying to write the same pre-rollback output fails repeatedly, the System will call the StartRollback on the Parameter Upgrade QA Device again, and subsequently calling the authenticated Write function on the Printer QA Device using this output.

28.3.3.4 Rollback

This state indicates a successful deletion (completion) of a transfer sequence. During this state, the Parameter Upgrader QA Device verifies the sequence data produced from StartRollBack has been correctly written to Printer QA Device, then rolls its count-remaining field to a previous value before the transfer request was issued.

28.3.4 Xfer Entry cache

The *Xfer Entry* data structure must allow for the following:

- Stores the *transfer state* and *sequence* data for a given *transfer sequence*.
- Store all data corresponding to a given transfer, to facilitate a *rollback* to the previous value before the transfer output was generated.

The *Xfer Entry cache* depth will depend on the QA Chip Logical Interface implementation. For some implementations a single *Xfer Entry* value will be saved. If the Parameter Upgrader QA Device has no powersafe storage of *Xfer Entry cache,* a power down will cause the erasure of the *Xfer Entry*

cache and the Parameter Upgrader QA Device will not be able to *rollback* to a pre-power-down value.

A dataset in the *Xfer Entry cache* will consist of the following:

- Information about the Printer QA Device:
  a. ChipId of the device.
  b. FieldNum of the M0 field (i.e what was being upgraded).
- Information about the Parameter Upgrader QA Device:
  a. ⋅FieldNum of the M0 field used to transfer the count-remaining from.
- Xfer State- indicating at which state the *transfer sequence* is. This will consist of:
  a. State definition which could be one of the following: - Xfer, StartRollBack and deleted (completed).
  b. The value of sequence data fields SEQ_1 and SEQ_2.

The Xfer Entry cache stores the FieldNum of the count-remaining field of the Parameter Upgrader QA Device.

28.3.4.1 Adding new dataset

A new dataset is added to Xfer Entry cache by the Xfer function.

There are three methods which can be used to add new dataset to the *Xfer Entry* cache. The methods have been listed below in the order of their priority:

1. *Replacing existing dataset in Xfer Entry cache with new dataset based on ChipId and FieldNum of the Ink QA Device in the new dataset.* A matching ChipId and FieldNum could be found because a previous *transfer* output corresponding to the dataset stored in the *Xfer Entry* cache has been correctly received and processed by the Parameter Upgrader QA Device, and a new transfer request for the same Printer QA Device, same field, has come through to the Parameter Upgrader QA Device.

2. *Replace existing dataset cache with new dataset based on the Xfer State.* If the Xfer State for a dataset indicates deleted (complete), then such a dataset will not be used for any further functions, and can be overwritten by a new dataset.

3. *Add new dataset to the end of the cache.* This will automatically delete the oldest dataset from the cache regardless of the *Xfer State*.

28.4   UPGRADING THE COUNT-REMAINING FIELD

This section is only applicable to the Parameter Upgrader QA Device.

The transfer of *count-remaining* is similar to transfer ink-remaining because both involve transferring of amounts. Therefore, this transfer uses the *XferAmount* function.

The *XferAmount* function performs additional checks when transferring *count-remaining*. This includes checking of the operating parameter field, associated with the *count-remaining*. They are as follows:

- The operating parameter *value* of the upgrading QA Device and the QA Device being upgraded must match.

- The operating parameter field (in both devices) must be upgradeable by one key only, and all other keys must have ReadOnly access. This *key* which has authenticated ReadWrite permission to the operating parameter field, must be different to the key that has authenticated Read Write permission to the count-remaining field.

- The data Type for the operating parameter field in the upgrading QA Device must match the data Type for the operating parameter field in the QA Device being upgraded.

28.5   NEW OPERATING PARAMETER FIELD INFORMATION

This section is only applicable to the Parameter Upgrader QA Device.

This field stores the operating parameter value that is copied from the Parameter Upgrader QA Device to the operating parameter field being updated in the Printer QA Device.

This field has a single key associated with it. This key has authenticated ReadWrite permission to this field and will be referred to as *write-parameter key*.

Table 299 shows the field information for the new operating parameter field in the Parameter Upgrader QA Device.

Attribute Name	Value	Explanation
Type	For e.g - TYPE_UPGRADE_PRINTSPEED_15[a]	Type describing the upgrade.
KeyNum	Slot number of the *write-parameter key*.	*Only the write-parameter key has authenticated ReadWrite access to this field.*
Non Auth RW Perm[b]	0	Non authenticated ReadWrite is not allowed to the field.
Auth RW Perm[c]	1	Authenticated (key based) ReadWrite access is allowed to the field.
KeyPerm	KeyPerms[KeyNum] = 0	KeyNum is the slot number of the *write-parameter* key which has *ReadWrite* permission to the field.
	KeyPerms[others= 0 ..7] = 0	All other keys have *ReadOnly* access.
End Pos		Set as required.

a. This is a sample type only and is not included in the Type Map in Appendix A.

b. Non authenticated Read Write permission.

c. Authenticated Read Write permission.

28.6   DIFFERENT TYPES OF TRANSFER

There can be three types of transfer:

- Parameter Transfer - This is transfer of an operating parameter value from a *Parameter Upgrader* QA Device to a Printer QA Device. This is performed when an upgradeable operating parameter is written (for the first time) or changed.

- Hierarchical refill - This is a transfer of count-remaining value from one *Parameter Upgrader Refill* QA Device to a *Parameter Upgrader* QA Device, where both QA Devices belong to the same OEM. This is typically performed when OEM divides the number of upgrades from one of its *Parameter Upgrader* QA Device to many of its *Parameter Upgrader* QA Devices.

- Peer to Peer refill - This is a transfer of count-remaining value from one *Parameter Upgrader Refill* QA Device to *Parameter Upgrader Refill* QA Device, where the QA Devices belong to different organisations, say ComCo and OEM. This is typically performed when ComCo divides number of upgrades from its *Parameter Upgrader* QA Device to several *Parameter Upgrader* QA Device belonging to several OEMs.

Transfer of count-remaining between peers, and hierarchical transfer of count-remaining, is similar to an ink transfer, but additional checks on the transfer request is performed when transferring count-remaining amounts. This is described in Section 28.4. 1.

Transfer of an operating parameter value decrements the count-remaining by 1, hence is different to a ink-transfer.

Figure 385 is a representation of various authorised upgrade paths in the printing system.

28.6.1   Hierarchical transfers

Referring to Figure 385, this transfer is typically performed when count-remaining amount is transferred from ComCo's Parameter Upgrader Refill QA Device to OEM's Parameter Upgrader Refill QA Device, or from QACo's Parameter Upgrader Refill QA Device to ComCo's Parameter Upgrader Refill QA Device.

This transfers are made using the *XferAmount* function (and not with the *XferField* described in Section 29.1), because count-remaining transfer is similar to fill/refilling of ink amounts, where ink amount is replaced by count-remaining amount.

28.6.1.1   *Keys and access permission*

We will explain this using a transfer from ComCo to OEM.

There is a *count-remaining* field associated with the ComCo's Parameter Upgrader Refill QA Device. This *count-remaining* field has *two keys* associated with:

- The *first key* transfers count-remaining to the device from another Parameter Upgrader Refill QA device(device is higher in the heirachy), fills/refills the device itself.

- The *second key* transfers count-remaining from it to other devices (which are lower in the heirachy), fills/refills other devices from it.

There is a *count-remaining* field associated with the OEM's Parameter Upgrader Refill QA Device. This *count-remaining* field has *a single key* associated with:

- This *key* transfers count-remaining to the device from another Parameter Upgrader Refilll QA device (which is higher or at the same level in the heirachy), fills/refills (upgrades) the device

5

itself, and additionally transfers count-remaining from it to other devices (which are lower in the heirachy), fills/refills (upgrades) other devices from it.

For a successful transfer of count-remaining from ComCo's refill device to an OEM's refill device, the ComCo's refill device and the OEM's refill device must share a *common* key or a *variant* key. This key is *fill/refill key* with respect to the OEM's refill device and it is the *transfer key* with respect

10 to the ComCo's refill device.

For a ComCo to successfully fill/refill its refill device from another refill device (which is higher in the heirachy possibly belonging to the QACo), the ComCo's refill device and the QACo's refill device must share a *common* key or a *variant* key. This key is *fill/refill key* with respect to the ComCo's refill device and it is the *transfer key* with respect to the QACo's refill device.

15 28.6.1.1.1   Count-remaining field information

Table 300 shows the field information for an $_{MO}$ field storing logical count-remaining amounts in the refill device, which has the ability to transfer down the heirachy.

Attribute Name	Value	Explanation
Type	TYPE_COUNT_REMAINING[a]	Type describes that the field is a count-remaining field.
KeyNum	Slot number of the *refill* key.	Only the *refill key* has authenticated *ReadWrite* access to this field.
Non Auth RW Perm[b]	0	*Non authenticated ReadWrite* is not allowed to the field.
Auth RW Perm[c]	1	*Authenticated (key based) ReadWrite access* is allowed to the field.
KeyPerm	KeyPerms[KeyNum] = 0	KeyNum is the slot number of the *refill* key, which has *ReadWrite* permission to the field.
	KeyPerms[Slot Num of *transfer* key] = 1	*Transfer key* can *decrement* the field.
	KeyPerms[others= 0 ..7(except transfer key)] = 0	All other keys have *ReadOnly* access.
End Pos	Set as required.	Depends on the amount of logical ink the

		device can store and storage resolution - i.e in picolitres or in microlitres.

a. Refer to Type Map in Appendix A for exact value.

b. Non authenticated Read Write permission.

c. Authenticated Read Write permission.

28.6.2   Peer to Peer transfer

Referring to Figure 385, this transfer is typically performed when count-remaining amount is transferred from OEM's Parameter Upgrader Refill QA Device to another Parameter Device Refill QA Device belonging to the same OEM.

28.6.2.1   Keys and access permission

There is an *count-remaining* field associated with the refill device. This *count-remaining* field has *a single key* associated with:

- This *key* transfers count-remaining amount to the device from another refill device (which is higher or at the same level in the heirachy), fills/refills (upgrades) the device itself, and additionally transfers ink from it to other devices (which are lower in the heirachy), fills/refills (upgrades) other devices from it.

This key is referred to as the *fill/refill key* and is used for *both fill/refill and transfer.* Hence, this *key* has both *ReadWrite* and *Decrement-Only* permission to the *count-remaining* field in the *refill device.*

28.6.2.1.1   Count-remaining field information

Table 301 shows the field information for an $_{M0}$ field storing logical count-remaining amounts in the refill device with the ability to transfer between peers.

Table 301. Field information for ink-remaining field for refill devices transferring between peers

Attribute Name	Value	Explanation
Type	TYPE_COUNT_REMAINING[a]	Type describes that the field is a count-remaining field.
KeyNum	Slot number of the *refill* key.	Only the *refill key* has authenticated *ReadWrite* access to this field.
Non Auth RW Perm[b]	0	*Non authenticated ReadWrite* is not allowed to the field.
Auth RW Perm[c]	1	*Authenticated (key based) ReadWrite access* is allowed to the field.
KeyPerm	KeyPerms[KeyNum] = 1	KeyNum is the slot number of the *refill* key,

		which has *ReadWrite* and *Decrement* permission to the field.
	KeyPerms[others= 0 ..7(except KeyNum)] = 0	All other keys have *ReadOnly* access.
End Pos	Set as required.	Depends on the amount of logical ink the device can store and storage resolution - i.e in picolitres or in microlitres.

a. Refer to Type Map in Appendix A for exact value.

b. Non authenticated Read Write permission.

c. Authenticated Read Write permission.

29 Functions

29.1 XFERFIELD

Input:	*KeyRef, $_{M0}$OfExternal, $_{M1}$OfExternal, ChipId, FieldNumL, FieldNumE, InputParameterCheck (Optional), R_E, SIG_E, R_{E2}*
Output:	*ResultFlag, Field data, R_{L2}, SIG_{out}*
Changes:	*$_{M0}$ and R_L*
Availablity:	*Parameter Upgrader QA Device*

29.1.1 Function description

The *XferField* is similar to the *XferAmount* function in that it produces data and signature for updating a given $_{M0}$ field. This data and signature when applied to the appropriate device through the *WriteFieldsAuth* function, will upgrade the *FieldNumE* ($_{M0}$ field) of a device to the same value as *FieldNumL* of the upgrading device.

The system calls the *XferField* function on the upgrade device with a certain *FieldNumL* to be transferred to the device being upgraded The *FieldNumE* is validated by the *XferField* function according to various rules as described in Section 29.1.4. If validation succeeds the *XferField* function produces the data and signature for subsequent passing into the *WriteFieldsAuth* function for the device being upgraded.

The transfer field output consists of the new data for the field being upgraded, field data of the two sequence fields, and a signature. When a transfer output is produced, the sequence field data in *SEQ_1* is decremented by 2 from the previous value *(as passed in with the input)*, and the sequence field data in *SEQ_2* is decremented by 1 from the previous value *(as passed in with the input)*.

Additional *InputParameterCheck* value must be provided for the parameters not included in the SIG_E, if the transmission between the System and Parameter Upgrader QA Device is error prone, and these errors are not corrected by the transimission protocol itself. *InputParameterCheck* is

SHA-1[FieldNumL | FieldNumE | XferValLength | XferVal], and is required to ensure the integrity of these parameters, when these inputs are received by the Parameter Upgrader QA Device. The *XferField* function must first calculate the *SHA-1[FieldNumL | FieldNumE]*, compare the calculated value to the value received (*InputParameterCheck*) and only if the values match act upon the inputs.

29.1.2   Input parameters

Table 302 describes each of the input parameters for XferField function.

Parameter	Description
KeyRef	*For common key input and output signature: KeyRef.keyNum* = Slot number of the key to be used for testing input signature and producing the output signature. SIG_E produced using $K_{KeyRef.keyNum}$ by the QA Device being upgraded. *SIGout* produced using $K_{KeyRef.keyNum}$ for delivery to the QA Device being upgraded. *KeyRef.useChipId* = 0
	For variant key input and output signatures: KeyRef.keyNum = Slot number of the key to be used for generating the variant key. SIG_E produced using a variant of $K_{KeyRef.keyNum}$ by the QA Device being upgraded. *SIGout* produced using a variant of $K_{KeyRef.keyNum}$ for delivery to the QA Device being upgraded. *KeyRef.useChipId* = 1 *KeyRef.chipId* = ChipId of the device which generated SIG_E and will receive *SIGout*.
M_0*OfExternal*	All 16 words of M_0 of the QA Device being upgraded
M_1*OfExternal*	All 16 words of M_1 of the QA Device being upgraded.
ChipId	ChipId of the QA Device being upgraded.
FieldNumL	M_0 field number of the local (updating) device. The data stored in this field will be copied from the upgrading device.
FieldNumE	M_0 field number of the QA Device being upgraded. This field will be updated to the value stored in *FieldNumL* within the upgrading device.
R_E	External random value used to verify input signature. This will be the *R* from the input signature generator (i.e device generating SIG_E). The input signal generator in this case, is the device being upgraded or a translation device.
R_{E2}	External random value used to produce output signature. This will be the R obtained by calling the *Random* function on the device which will receive the SIG_{out} from the *XferField* function. The device receiving the SIG_{out} in this case, is the device being upgraded or a translation device.

869

SIG_E	External signature required for authenticating input data. The input data in this case, is the output from the *Read* function performed on the device being upgraded. A correct SIG_E = SIG$_{KeyRef}$(Data | R$_E$ | R$_L$).

29.1.2.1 *Input signature verification data format*

Refer to Section 27.1.2.1.

29.1.3 Output parameters

5

Table 303 describes each of the output parameters for XferField function.

Parameter	Description
ResultFlag	Indicates whether the function completed successfully or not. If it did not complete successfully, the reason for the failure is returned here. See Section 12.1, Table 292 and Table 303.
FieldSelect	Selection of fields to be written In this case the bit corresponding to *SEQ_1* , *SEQ_2* and to *FieldNumE* are set to 1. All other bits are set to 0.
FieldVal	Updated data words for *sequence data* field *and FieldNumE* for QA Device being upgraded. Starts with LSW of lower field. This must be passed as input to the *WriteFieldsAuth* function of the QA Device being upgraded.
R_{L2}	Internal random value required to generate output signature This must be passed as input to the *WriteFieldsAuth* function or *Translate* function of the QA Device being upgraded.
SIG_{out}	Output signature which must be passed as an input to the *WriteFieldsAuth* function or *Translate* function of the QA Device being upgraded. SIG$_{out}$ = SIG$_{KeyRef}$(data | R$_{L2}$ | R$_{E2}$) as per Figure 373

10

Table 303. Result Flag definitions for XferField

ReultFlag Definition	Description
CountRemainingFieldInvalid	The count- remaining field in Upgrading QA Device is invalid.
FieldNumEKeyPermInvalid	The upgrade field in the QA Device being upgraded doesn't have the correct authenticated permission.
NoUpgradesRemaining	The count-remaining field assocaited with the upgrade field in the Upgrading QA Device doesn't have any more upgrades left.

29.1.3.1 Output signature generation data format

Refer to Section 27.1.3.1.

29.1.4   Function sequence

The *XferField* command is illustrated by the following pseudocode:

Accept input parameters-KeyRef, M0OfExternal, M1OfExternal, ChipId, FieldNumL, FieldNumE, R_E, SIG_E, R_{E2}

```
#Generate message for passing into ValidateKeyRefAndSignature
function
data ← (RWSense|MSelect|KeyIdSelect|ChipId|WordSelect|M0|M1)
        # Refer to Figure 382.

----------------------------------------------------------------

# Validate KeyRef, and then verify signature
ResultFlag = ValidateKeyRefAndSignature(KeyRef,data,R_E,R_L)
If (ResultFlag ≠ Pass)
   Output ResultFlag
   Return
EndIf
----------------------------------------------------------------

# Validatate FieldNumE
# FieldNumE is present in the device being upgraded
PresentFlagFieldNumE ← GetFieldPresent(M1OfExternal,FieldNumE)

# Check FieldNumE present flag
If(PresentFlagFieldNumE ≠ 1)
```

```
          ResultFlag ← FieldNumEInvalid

          Output ResultFlag

          Return

      EndIf
```

5

```
      -----------------------------------------------------------------

      # Check Seq fields exist and get their Field Number

      # Get Seqdata field SEQ_1 for the device being upgraded

      XferSEQ_1FieldNum← GetFieldNum(M1OfExternal, SEQ_1)
```

10

```
      # Check if the Seqdata field SEQ_1 is valid

      If(XferSEQ_1FieldNum invalid)

          ResultFlag ← SeqFieldInvalid

          Output ResultFlag
```

15

```
          Return

      EndIf

      # Get Seqdata field SEQ_2 for the device being upgraded

      XferSEQ_2FieldNum← GetFieldNum(M1OfExternal, SEQ_2)
```

20

```
      # Check if the Seqdata field SEQ_2 is valid

      If(XferSEQ_2FieldNum invalid)

          ResultFlag ← SeqFieldInvalid

          Output ResultFlag

          Return
```

25

```
      EndIf

      -----------------------------------------------------------------

      ---------------

      #Check write permission for FieldNumE
```

30

```
      PermOKFieldNumE ← CheckFieldNumEPerm(M1OfExternal,FieldNumE)

      If(PermOKFieldNumE ≠ 1)

          ResultFlag ← FieldNumEWritePermInvalid

          Output ResultFlag

          Return
```

35

```
      EndIf
```

```
---------------------------------------------------------------------
---------------

#Check that both SeqData fields have Decrement-Only permission
with the same key
#that has write permission on FieldNumE
PermOKXferSeqData ← CheckSeqDataFieldPerms(M1OfExternal,
                            XferSEQ_1FieldNum,
XferSEQ_2FieldNum,FieldNumE)
If(PermOKXferSeqData ≠ 1)
    ResultFlag ← SeqWritePermInvalid
    Output ResultFlag
    Return
EndIf
---------------------------------------------------------------------
---------------


# Get SeqData SEQ_1 data from device being upgraded
GetFieldDataWords(XferSEQ_1FieldNum,

    XferSEQ_1DataFromDevice,M0OfExternal,M1OfExternal)


# Get SeqData SEQ_2 data from device being upgraded
GetFieldDataWords(XferSEQ_2FieldNum,
                        XferSEQ_2DataFromDevice,
M0OfExternal,M1OfExternal)


---------------------------------------------------------------------
# FieldNumL(upgrade value)is a valid field in the upgrading device
PresentFlagFieldNumL ← GetFieldPresent(M1,FieldNumL)
If(PresentFlagFieldNumL ≠ 1)
    ResultFlag ← FieldNumLInvalid
    Output ResultFlag
    Return
EndIf
---------------------------------------------------------------------
#Get the CountRemaining field associated with the upgrade value
field
```

```
                     # The CountRemaining field is the next higher field from the
                     upgrade value field
                     FieldNumCountRemaining← FieldNumL + 1


5                    # FieldNumCountRemaining is a valid field in the upgrading device
                     PresentFlagFieldNumCountRemaining
                     ← GetFieldPresent(M1,FieldNumCountRemaining)
                     If(PresentFlagFieldNumCountRemaining ≠ 1)
                         ResultFlag ← CountRemainingFieldInvalid
10                       Output ResultFlag
                         Return
                     EndIf
                     ------------------------------------------------------------------
                     #Check permission for upgrade value field. Only one key (different
15                   # from KeRef.keyNum) has write permissions to the field and no key
                     has decrement permissions.
                     CheckOK ← CheckUpgradeKeyForField(FieldNumL,M1,KeyRef)
                     If(CheckOK ≠ 1)
                         ResultFlag ← FieldNumEKeyPermInvalid
20                       Output ResultFlag
                         Return
                     EndIf
                     ------------------------------------------------------------------
                     #Find the type attribute for FieldNumE
25                   TypeFieldNumE ← FindFieldNumType(M1OfExternal,FieldNumE)
                     #Find the type attribute for FieldNumL (upgrade value)
                     TypeFieldNumL ← FindFieldNumType(M1,FieldNumL)

                     If(TypeFieldNumE ≠ TypeFieldNumL)
30                       ResultFlag ← TypeMismatch
                         Output ResultFlag
                         Return
                     EndIf
                     ------------------------------------------------------------------
35
```

```
        # Check permissions for CountRemaining field
        # Check upgrades are available in the CountRemaining field of the
        # upgrading device i.e value of CountRemaining is non-zero
        positive number
5       CountRemainingOK ←CheckCountRemaining(FieldNumCountRemaining, M0,
        M1)
        If(CountRemainingOK ≠ 1)
            ResultFlag ← NoUpgradesRemaining
            Output ResultFlag
10          Return
        EndIf
        -----------------------------------------------------------------


        #Get the size of the FieldNumL (upgrade value)
15      If(FieldNumL = 0)
            FieldSizeOfFieldNumL← MaxWordInM- M1[FieldNumL].EndPos
        Else
            FieldSizeOfFieldNumL← M1[FieldNumL-1].EndPos-
        M1[FieldNumL].EndPos
20      EndIf



        #Get the size of the FieldNumE (field being updated)
        If(FieldNumL = 0)
25          FieldSizeOfFieldNumE←MaxWordInM- M1OfExternal[FieldNumE -
        1].EndPos
        Else
            FieldSizeOfFieldNumE← M1OfExternal[FieldNumE-1].EndPos
                                    - M1OfExternal[FieldNumL].EndPos
30      EndIf

        # Check whether the device being upgraded can hold the upgrade
        value from
        # FieldNumL
35      If(FieldSizeOfFieldNumE < FieldSizeOfFieldNumL)
            ResultFlag ← FieldNumESizeInsufficient
            Output ResultFlag
```

```
        Return

        EndIf

        --------------------------------------------------------------------

        # All checks complete .....



        # Generate Seqdata for SEQ_1 and SEQ_2 fields

        XferSEQ_1DataToDevice = XferSEQ_1DataFromDevice - 2

        XferSEQ_2DataToDevice = XferSEQ_2DataFromDevice - 1



        # Add DataSet to Xfer Entry Cache

        AddDataSetToXferEntryCache(ChipId,FieldNumE, FieldNumL,

        XferSEQ_1DataFromDevice, XferSEQ_2DataFromDevice)



        #Decrement CountRemaining field by one

        DecrementField(FieldNumCountRemaining,M0)



        #Get the upgrade value words from FieldNumE of the upgrading
        device

        GetFieldDataWords(FieldNumL,UpgradeValue,M0,M1)



        #Generate new field data words for FieldNumE. The upgrade value is
        copied to

        FieldDataE

        FieldDataE← UpgradeValue



        # Generate FieldSelect and FieldVal for SeqData field SEQ_1, SEQ_2
        and

        # FieldDataE...

        CurrentFieldSelect← 0

        FieldVal ← 0

        GenerateFieldSelectAndFieldVal(FieldNumE, FieldDataE,

        XferSEQ_1FieldNum, XferSEQ_1DataToDevice,XferSEQ_2FieldNum,

        XferSEQ_2DataToDevice,

        FieldSelect,FieldVal)



        #Generate message for passing into GenerateSignature function
```

```
data ← (RWSense|FieldSelect|ChipId|FieldVal)# Refer to Figure 373.
#Create output signature for FieldNumE
SIG_out← GenerateSignature(KeyRef,data,R_L2,R_E2)
Update R_L2 to R_L3
ResultFlag ← Pass
Output ResultFlag, FieldSelect,FieldVal, R_L2 ,SIG_out
Return
EndIf
```

29.1.4.1 CountRemainingOK

CheckCountRemainingFieldNumL(FieldNumCountRemaining,M1,M0)

This functions checks permissions for CountRemaining field and also checks that upgrades are available in the CountRemaining field of the upgrading device.

```
AuthRW ← M1[FieldNumCountRemaining].AuthRW
NonAuthRW ← M1[FieldNumCountRemaining].NonAuthRW
DOForKeys ← M1[FieldNumCountRemaining].DOForKeys[KeyNum]
Type ← M1[FieldNumCountRemaining].Type
If(AuthRW = 1 ∧ NonAuthRW = 0 ∧ (DOForKeys = 1∧ (Type =
TYPE_COUNT_REMAINING)
    PermOK ←1
Else
    PermOK ← 0
    Return PermOK
EndIf
#Get the count-remaining value from the upgrading device
GetFieldDataWords(FieldNumCountRemaining,CountRemainingValue,M0,M1
)
If(CountRemainingValue <= 0)
    PermOK ← 0
    Return PermOK
EndIf
PermOK ← 1
Return PermOK
```

29.2 ROLLBACKFIELD

> **Input:** KeyRef, $_{M0}$OfExternal, $_{M1}$OfExternal, ChipId, FieldNumL, FieldNumE, InputParameterCheck (optional), R_E, SIG_E
>
> **Output:** ResultFlag
>
> **Changes:** $_{M0}$ and R_L
>
> **Availablity:** Parameter Upgrader QA Device

29.2.1 Function description

The *RollBackField* function is very similar to the *RollBackAmount* function, the only difference being that the *RollBackField* function adjusts the value of the *count-remaining* field associated with the *upgrade value* field of the upgrading device, instead of the *upgrade value* field itself. A successful rollback, increments the *count-remaining* by 1.

The Parameter Upgrader QA Device checks that the Printer QA Device didn't actually receive the transfer message correctly, by comparing the sequence data field values read from the device with the values stored in the *Xfer Entry* cache. The sequence data field values read must match what was previously written using the *StartRollBack* function. After all checks are fulfilled, the Parameter Upgrader QA Device adjusts its *FieldNumL*.

Additional *InputParameterCheck* value must be provided for the parameters not included in the SIG_E, if the transmission between the System and Parameter Upgrader QA Device is error prone, and these errors are not corrected by the transimission protocol itself. *InputParameterCheck* is *SHA-1[FieldNumL | FieldNumE]*, and is required to ensure the integrity of these parameters, when these inputs are received by the Parameter Upgrader QA Device.

The *RollBackField* function must first calculate the *SHA-1[FieldNumL | FieldNumE]*, compare the calculated value to the value received (*InputParameterCheck*) and only if the values match act upon the inputs.

29.2.2 Input parameters

Table 305 describes each of the input parameters for RollBackField function.

Parameter	Description
KeyRef	For common key input signature: KeyRef.keyNum = Slot number of the key to be used for testing input signature. SIG_E produced using $K_{KeyRef.keyNum}$ by the QA Device being upgraded. KeyRef.useChipId = 0
	For variant key input signature: KeyRef.keyNum = Slot number of the key to be used for generating the variant key. SIG_E produced using a variant of $K_{KeyRef.keyNum}$ by the QA Device being upgraded. KeyRef.useChipId = 1 KeyRef.chipId = ChipId of the device which

	generated SIG_E.
$_{M0}OfExternal$	16 words of $_{M0}$ of the QA Device being upgraded which failed to upgrade.
$_{M1}OfExternal$	16 words of $_{M1}$ of the QA Device being upgraded which failed to upgrade.
$ChipId$	ChipId of the QA Device being upgraded which failed to upgrade.
$FieldNumL$	$_{M0}$ field number of the local (upgrading) device whose value could not be copied to the device being upgraded.
$FieldNumE$	$_{M0}$ field number of the QA Device being upgraded to which the upgrade value in $FieldNumL$ couldn't be copied.
R_E	External random value used to verify input signature. This will be the R from the input signature generator (i.e device generating SIG_E). The input signal generator in this case, is the device which failed to upgrade or a translation device.
SIG_E	External signature required for authenticating input data. The input data in this case, is the output from the $Read$ function performed on the device which failed to upgrade. A correct $SIG_E = SIG_{KeyRef}(Data \mid R_E \mid R_L)$.

29.2.2.1 Input signature generation data format

Refer to Section 27.1.2.1 for details.

29.2.3 Output parameters

5

Table 306 describes each of the output parameters for RollBackField.

Parameter	Description
$ResultFlag$	Indicates whether the function completed successfully or not. If it did not complete successfully, the reason for the failure is returned here. See Section 12.1, Table 292, Table 304 and Table 295.

29.2.4 Function sequence

10

The $RollBackField$ command is illustrated by the following pseudocode:

```
Accept input parameters-KeyRef, M0OfExternal, M1OfExternal,
ChipId, FieldNumL, FieldNumE, R_E,SIG_E
```

#Generate message for passing into GenerateSignature function

```
data ← (RWSense|MSelect|KeyIdSelect|ChipId|WordSelect|M0|M1)
        # Refer to Figure 382.
---------------------------------------------------------------

# Validate KeyRef, and then verify signature
ResultFlag = ValidateKeyRefAndSignature(KeyRef,data,$R_E$,$R_L$)
If (ResultFlag ≠ Pass)
   Output ResultFlag
   Return
EndIf
---------------------------------------------------------------

# Check Seq fields exist and get their Field Number
# Get Seqdata field SEQ_1 num for the device being upgraded
XferSEQ_1FieldNum← GetFieldNum(M1OfExternal, SEQ_1)


# Check if the Seqdata field SEQ_1 is valid
If(XferSEQ_1FieldNum invalid)
   ResultFlag ← SeqFieldInvalid
   Output ResultFlag
   Return
EndIf
# Get Seqdata field SEQ_2 num for the device being upgraded
XferSEQ_2FieldNum← GetFieldNum(M1OfExternal, SEQ_2)

# Check if the Seqdata field SEQ_2 is valid
If(XferSEQ_2FieldNum invalid)
   ResultFlag ← SeqFieldInvalid
   Output ResultFlag
   Return
EndIf
---------------------------------------------------------------

# Get SeqData SEQ_1 data from device being upgraded
GetFieldDataWords(XferSEQ_1FieldNum,

   XferSEQ_1DataFromDevice,M0OfExternal,M1OfExternal)
```

5

10

15

20

25

30

35

```
      # Get SeqData SEQ_2 data from device being upgraded
      GetFieldDataWords(XferSEQ_2FieldNum,
                        XferSEQ_2DataFromDevice,
5     M0OfExternal,M1OfExternal)


      # Generate Seqdata for SEQ_1 and SEQ_2 fields with the data that
      is read
      XferSEQ_1Data = XferSEQ_1DataFromDevice + 1
10    XferSEQ_2Data = XferSEQ_2DataFromDevice + 2



      # Check Xfer Entry in cache is correct - dataset exists, Field
      data
15    # and sequence field data matches and Xfer State is correct
      XferEntryOK ← CheckEntry(ChipId, FieldNumE, FieldNumL,
                        XferSEQ_1Data, XferSEQ_2Data)


      If( XferEntryOK= 0)
20        ResultFlag ← RollBackInvalid
          Output ResultFlag
          Return
      EndIf


25    # Increment associated CountRemaining by 1
      IncrementCountRemaining(FieldNumCountRemaining)
      # Update XferState in DataSet to complete/deleted
      UpdateXferStateToComplete(ChipId,FieldNumE)
      ResultFlag ← Pass
30    Output ResultFlag
      Return
```

EXAMPLE SEQUENCE OF OPERATIONS

30   Concepts

The QA Chip Logical Interface interface devices do not initiate any activities themselves. Instead
35 the System reads data and signature from various untrusted devices, and sends the data and
signature to a trusted device for validation of signature, and then uses the data to perform
operations required for printing, refilling, upgrading and key replacement. The system will therefore

881

be responsible for performing the functional sequences required for printing, refilling, upgrading and key replacement. It formats all input parameters required for a particular function, then calls the function with the input parameters on the appropriate QA Chip Logical Interface instance, and then processes/stores the output parameters from the function appropriately.

5   Validation of signatures is achieved by either of the following schemes:

- *Direct* - the signature produced by an untrusted device is directly passed in for validation to the trusted device. The direct validation requires the untrusted device to share a *common* key or a *variant* key with the trusted device. Refer to Section 7 for further details on *common* and *variant* keys.

10   - *Translation* - the signature produced by an untrusted is first validated by the translating device, and a new signature of the read data is produced by the translation device for validation by the trusted device. Several translation device may be chained together - the first translation device validates the signature from the untrusted device, and the last translation device produces the final signature for validation by the trusted device. The translation device

15   must share a *common* key or a *variant* key with the trusted/untrusted device and among themselves, if several translation devices are chained together for signature validation.

30.1   REPRESENTATION

Each functional sequence consists of the following devices (refer to Section 4.3):

- *System*.

20   - A *trusted QA Device* - which may be a *system trusted QA Device*, or an *Parameter Upgrader QA Device*, or a *Ink Refill QA Device,* or a *Key Programmer QA Device* depending on the function performed. This device is referred to as device A.

- An *untrusted QA Device* - which may be a *Printer QA Device*, or an *Ink QA Device*. This device is referred to as device B.

25   - A *translation QA Device* will be used if a translation scheme is used to validate signatures. This device is referred to as device C.

The command sequence produced by the system for further sequences will be documented as shown in Table 307.

Table 307. Command sequence representation

30

Sequence No	Function	Parameters
Sequence order	Device.FunctionName	Input Parameters and their values.
		Output parameters and their description.

882

Therefore, a typical *direct signature* validation sequence can be represented by Figure 386 and Table 308.

For a direct signature to be used, A and B must share a *common* or a *variant* key
5   i.e B.K_{n1} = A.K_{n2} or B.K_{n1} = FormKeyVariant(A.K_{n2}, B.ChipId).

Table 308. Command sequence for direct signature validation

Sequence No	Function	Parameters
1	A.*Random*	None
		R_A=RL
2	B.*Read*	KeyRef = n1, SigOnly = 0, MSelect = Any one M, KeyIdSelect = 0, WordSelectForDesiredM = Any one word in the selected M, RE = R_A
		If ResultFlag = Pass then MWords = SelectedWordsOfSelectedMs as per input [MSelect] and [WordSelectForDesiredM], R_B = R_L, SIG_B = SIGout Refer to Section 15.3.1.
3	A.*Test*	KeyRef = n2, DataLength = Length of MWords in words preformatted as per Section 16.1, Data = MWords preformatted as per Section 16.1, RE =R_B, SIGE = SIG_B
		ResultFlag = Pass/Fail

10

A typical *signature validation using translation* can be represented by Figure 387 and Table 309.

For validating signatures using translation:

- A and C must share a *common* or a *variant* key

  i.e $C.K_{n3} = A.K_{n2}$ or $C.K_{n3} = FormKeyVariant(A.K_{n2}, C.ChipId)$.

- B and C must share a *common* or a *variant* key

  i.e $C.K_{n2} = B.K_{n1}$ or $B.K_{n1} = FormKeyVariant(C.K_{n2}, B.ChipId)$.

Table 309. Command sequence for signature validation using translation

Sequence No	Function	Parameters
1	C.Random	None
		$R_C = RL$
2	B.Read	KeyRef = n1, SigOnly = 1 or 0, MSelect = any, KeyIdSelect = any, WordSelectForDesiredM = any, RE= R_C
		If ResultFlag = Pass then MWords = SelectedWordsOfSelectedMs as per input [MSelect] and [WordSelectForDesiredM], $R_B = R_L$, $SIG_B = SIGout$ Refer to Section 15.3.1.
3	A.Random	None
		$R_A = RL$
4	C.Translate	InputKeyRef =n2, DataLength = Length of MWords in words preformatted as per Section 17.1, Data = MWords preformatted as per Section 17.1, RE= R_B, SIGE = SIG_B, OutputKeyRef = n3, RE2 = R_A
		If ResultFlag = Pass then $R_{C1}= R_{L2}$, $SIG_C= SIGOut$ Refer to Section 15.3.1
5	A.Test	KeyRef = n2, DataLength = Length of MWords in words preformatted as per Section 16.1, Data = MWords preformatted as per Section 16.1, RE =R_{C1}, SIGE = SIG_C
		ResultFlag = Pass/Fail

31   In field use

This section covers functional sequences for printer and ink QA Devices, as they perform their usual function of printing.

31.1 STARTUP SEQUENCE

At startup of any operation (a printer startup or an upgrade startup), the system determines the properties of each QA Device it is going to communicate with. These properties are:

5 • Software version of the QA Device. This includes *SoftwareReleaseIdMajor* and *SoftwareReleaseIdMinor*. The *SoftwareReleaseIdMajor* identifies the functions available in the QA Device. Refer to Section 13.2 for details.

• The number of memory vectors in the QA Device.

• The number of keys in the QA Device.

10 • The ChipId of the QA Device.

The properties allow the system to determine which functions are available in a given QA Device, as well as the value of input parameters required to communicate with the QA Device.

Table 310 shows the startup sequence.

Table 310. Startup command sequence

15

Sequence No	Function	Command
1	B.GetInfo	None
		Major release identifier of the QA Device = SoftwareReleaseIdMajor, Minor release identifier of the QA Device= SoftwareReleaseIdMinor, Number of memory vectors in the QA Device= NumVectors, Number of keys in the QA Device= NumKeys, Id of the QA Device = ChipId 0 = VarDataLen No VarData in case of an ink or printer QA Device

31.1.1 Clearing the preauthorisation field

Preauthorisation of ink is one of the schemes that a printer may use to decrement logical ink as physical ink is used. This is discussed in details in Section 31.4.3.

20 If the printer uses preauthorisation, the system must read the preauthorisation field at startup. If the preauthorisation field is not clear, then the system must apply (decrement) the preauth amount to the corresponding ink field, by performing a non-authenticated write of the decremented amount to the appropriate ink field, and then clear the preauthorisation field by performing an authenticated write to the preauthorisation field.

25 ## 31.2 PRESENCE ONLY AUTHENTICATION

The purpose of presence only authentication is to determine whether the printer should or shouldn't work with the ink cartridge.

31.2.1 Without data interpretation

This sequence is performed when the printer authenticates the ink cartridge. The authentication consists of verifying a signature generated by the untrusted ink QA Device (in the ink cartridge) using the system's trusted QA Device.

For signature to be valid, the trusted QA Device (A) and the untrusted ink QA Device (B) must share a *common* or a *variant* key i.e $B.K_{n1} = A.K_{n2}$ or $B.K_{n1} = FormKeyVariant(A.K_{n2}, B.ChipId)$.

A single word of a single M is read because the system is only interested in the validity of signature for a given data.

If the printer wants to verify the signature and doesn't require any data from the ink cartridge (because it is cached in the printer), then the printer calls the *Read* function with *SigOnly* set to 1. The *Read* returns only the signature of the data as requested by the input parameters. The printer then sends its cached data and signature (from the *Read* function) to its trusted QA Device for verification. The printer may use this signature verification scheme if it has read the data previously from the ink QA Device, and the printer knows that the data in the ink QA Device has not changed from value that was read earlier by the printer.

Table 311 shows the command sequence for performing presence only authentication requiring both data and signature.

Seq No	Function	Parameters
1	A.Random	None
		R_A = RL
2	B.Read	KeyRef = n1, SigOnly = 0, MSelect = Any one M, KeyIdSelect = 0, WordSelectForDesiredM = Any one word in the selected M, RE= R_A
		If ResultFlag = Pass then MWords = SelectedWordsOfSelectedMs as per input [MSelect] and [WordSelectForDesiredM], R_B = R_L, SIG_B = SIGout Refer to Section 15.3.1.
3	A.Test	KeyRef = n2, DataLength = Length of MWords in words preformatted as per Section 16.1, Data = MWords preformatted as per Section 16.1, RE =R_B, SIGE = SIG_B
		ResultFlag = Pass/Fail

31.2.2 With data interpretation

This sequence is performed when the printer reads the relevant data from the untrusted QA Device in the ink cartridge. The system validates the signature from the external ink QA Device, and then uses this data for further processing.

For signature to be valid, the trusted QA Device (A) and the untrusted QA Device (B) must share a common or a variant key i.e $B.K_{n1} = A.K_{n2}$ or $B.K_{n1} = FormKeyVariant(A.K_{n2}, B.ChipId)$.

The data read assists the printer to determine the following before printing can commence:

- Which fields in $_{M0}$ store logical ink amounts in the ink QA Device.
5 - The size of the ink fields in the ink QA Device. Refer to Section 8.1.1.1.
- The type of ink.
- The amount of ink in the field.

Table 312 shows the command sequence for performing presence only authentication (with data interpretation).

10

Seq No	Function	Parameters
1	A.Random	None
		$R_A = RL$
2	B.Read	KeyRef = n1, SigOnly = 0, MSelect = 0x03(indicates M0 and M1), KeyIdSelect = 0xFF (Read all KeyIds), WordSelectForDesiredM (for $_{M0}$)= 0xFFFF (Read all 16 $_{M0}$words), WordSelectForDesiredM (for $_{M1}$)= 0xFFFF(Read all 16 $_{M1}$words), RE= R_A
		If ResultFlag = Pass then MWords = SelectedWordsOfSelectedMs as per input [MSelect] and [WordSelectForDesiredM], All 16 words of $_{M0}$ and $_{M1}$. R_B = RL SIG$_B$ = SIGout Refer to Section 15.3.1
3	A.Test	Input Key = n2, DataLength = Length of MWords in words preformatted as per Section 16.1, Data = MWords preformatted as per Section 16.1, RE =R_B, SIGE = SIG$_B$
		ResultFlag = Pass/Fail

31.2.2.1 Locating ink fields and determining ink amounts remaining

Before printing can commence, the printer must determine the ink fields in the ink cartridge so that it
15 can decrement these fields with the physical use of ink. The printer must also verify that the ink in the ink cartridge is suitable for use by the printer.

This process requires reading data from the ink QA Device and then comparing the data to what is required. To perform the comparison the printer must store a list for each ink it uses.

The ink list must consist of the following:

20 - Ink Id - A identifier for the ink
- KeyId - The KeyId of the key used to fill/refill this ink.
- Type - This is the type attribute of the ink.

The ink list stored in the printer is shown in Table 313.

Ink Id	KeyId	Type
1- represents black ink	1- represents KeyId of Network_OEM_InkFill/Refill Key[b]	0x55 TYPE_REGULAR_BLACK_INK[a]
2- represents cyan ink	1- represents KeyId of Network_OEM_InkFill/Refill Key[b]	0x9F TYPE_HIGHQUALITY_CYAN_INK[a]
3- represents magenta ink	1- represents KeyId of Network_OEM_InkFill/Refill Key[b]	0x9A TYPE_HIGHQUALITY_MAGENTA_INK[a]
4- represents yellow ink	1- represents KeyId of Network_OEM_InkFill/Refill Key[b]	0x9C TYPE_HIGHQUALITY_YELLOW_INK[a]

5   a. These Types are only used as an example.

       b. These KeyIds are only used as an example.

The printer will perform a *Read* of the ink QA Device's M0, M1 and KeyIds to determine the
following:

- The correct ink field ($_{M0}$ field) in the ink QA Device.

10   - The amount of ink-remaining in the field.

The ink QA Device's M1 and KeyId helps the printer determine the location of the ink field and ink
QA Device's M0 and M1 helps determine the amount of ink-remaining in the field.

31.2.2.2 FieldNum FindFieldNum(keyIdRequired, typeRequired)

This function returns a *FieldNum* of an M0 field, whose authenticated ReadWrite access key's

15   *KeyId* is *keyIdRequired*, and whose *Type* attribute matches *typeRequired*. If no matching field is
found it returns a *FieldNum = 255*. This function must be available in the printer system so that it
can determine the ink field required by it.

The function sequence is described below.

```
         # Get total number of fields in the ink QA Device
20       FieldSize[16]  ← 0 # Array to hold FieldSize assuming there are 16
         fields
         NumFields← FindNumberOfFieldsInM0(M1,FieldSize)  # Refer to
         Section 19.4.1.
```

```
# Loop through KeyIds read assuming all KeyIds have been read from
ink QA Device
For i ← 0 to 7
    #Check if KeyId read matches


    If(KeyIdᵢ = keyIdRequired # Matching KeyId found
        KeyNum ← i            # Get the KeyNum of the matching KeyId


        # Now look through the field to check which field has
        #write permissions with this KeyNum


        For j ← 0 to NumOfFields
            AuthRW ← M1[j].AuthRW # Isolate AuthRW for field
            # Check authenticated write is allowed to the field
            If(AuthRW = 1)
                KeyNumⱼ← Mₗ[j].KeyNum # Isolate KeyNum of the field
                Typej ←Mₗ[j].Type #Islotate Type attribute of the field
                # Check if Key is write key for the field and type of
Ink Id#2
                If(KeyNum = KeyNumⱼ) ∧ (Typeⱼ = typeRequired)
                    FieldNum ← j
                    return FieldNum
                EndIf
            EndIf
        EndFor # Loop through to next field
        FieldNum ← 255 # Error - no field found
        return FieldNum
    EndIf
EndFor # Loop through to next KeyId
```

For e.g if the printer wants to find an ink field that matches Ink Id#2 (from Table 313) in the ink QA Device, it must call the function *FindFieldNum* with *keyIdRequired* = KeyId of Network_OEM_InkFill/Refill Key and *typeRequired* = TYPE_HIGHQUALITY_CYAN_INK.

31.2.2.3 Ink-remaining amount

This can be determined by using the function *GetFieldDataWords(FieldNum,FieldData[], M0,M1)* described in Section 27.1.4.14. *FieldNum* must be set to the value returned from function in Section 31.2.2.2. FieldData returns the ink-remaining amount.

The function *GetFieldDataWords(FieldNum,FieldData[], M0,M1)* must be implemented in the printer system.

31.3   PRESENCE ONLY AUTHENTICATION THROUGH THE TRANSLATE FUNCTION

This sequence is performed when the printer reads the data from the untrusted ink QA Device in the ink cartridge but uses a translating QA Device to indirectly validate the read data. The translating QA Device validates the signature using the key it shares with the untrusted QA Device, and then signs the data using the key it shares with the trusted QA Device. The trusted QA Device then validates the signature produced by the translating QA Device.

For validating signatures using translation:

- A and C must share a *common* or a *variant* key

  i.e $C.K_{n3} = A.K_{n2}$ or $C.K_{n3} = FormKeyVariant(A.K_{n2}, C.ChipId)$.

- B and C must share a *common* or a *variant* key

  i.e $C.K_{n2} = B.K_{n1}$ or $B.K_{n1} = FormKeyVariant(C.K_{n2}, B.ChipId)$.

Table 314 shows a command sequence for presence only authentication using translation

Seq No	Function	Parameters
1	C.Random	None
		$R_C = RL$
2	B.Read	KeyRef = n1, SigOnly = 1 or 0, MSelect = any M, KeyIdSelect = 0, WordSelectForDesiredM = any, RE= R_C
		If ResultFlag = Pass then MWords = SelectedWordsOfSelectedMs as per input [MSelect] and [WordSelectForDesiredM], $R_B = R_L$, $SIG_B = $ SIGout Refer to Section 15.3.1
3	A.Random	None
		$R_A = RL$
4	C.Translate	InputKeyRef =n2, DataLength = Length of MWords in words preformatted as per Section 17.1, Data = MWords preformatted as per Section 17.1, RE= R_B, SIGE = SIG_B, OutputKeyRef = n3, RE2 = R_A
		If ResultFlag = Pass then $R_{C1}=RL1$, $SIG_C=$ SIGOut Refer to Section 15.3.1

5	A.Test	KeyRef = n2, DataLength = Length of MWords in words preformatted as per Section 16.1, Data = MWords preformatted as per Section 16.1, RE =R_{C1}, SIGE = SIG_C
		ResultFlag = Pass/Fail

31.4 UPDATING THE INK-REMAINING

This sequence is performed when the printer is printing. The ink QA Device holds the logical amount of ink-remaining corresponding to the physical ink left in the cartridge. This logical ink amount must decrease, as physical ink from the ink cartridge is used for printing.

31.4.1 Sequence of update

The primary question is *when* to deduct the logical ink amount - before or after the physical ink is used.

a. *Print first (use physical ink) and then update the logical ink.* If the power is cut off after a physical print and before a logical update, then the logical update is not performed. Therefore, the logical ink-remaining is more than the physical ink-remaining. Performing repeated power cuts will increase the differential amount, and finally any physical ink could be used to refill the QA Device.

b. *Update the logical ink and then print (use physical ink).* This is better than (*a*) because other physical inks cannot be used. However, if a problem occurs during printing, after the logical amount has already been deducted, there will be a disparity between logical and physical amounts. This might result in the printer not printing even if physical ink is present in the ink cartridge. The amount of disparity can be reduced by increasing the frequency of updating logical ink i.e update after each line instead of after each page.

c. *Preauthorise logical ink.* Preauthorise certain amount of ink (depends on the frequency of logical updates) before print and clear it at the end of printing. If power is cut off after a page is printed, then on start up, the printer reads the preauthorisation field, if it has not been cleared, it applies the preauth amount to the ink-remaining amount, and then clears the preauthorisation field.

31.4.2 Basic update

Some printers may use one of methods described in Section 31.4.1 (a) or (b) to update logical ink amounts in the ink QA Device. This method of updating the ink is termed as a basic update. The decremented amount is written to the appropriate ink field (which has been previously determined using Section 31.2.2) in $_{M0}$. The printer verifies the write, by reading the signature of the written data, then passing it to the *Test* function of the trusted QA Device.

For signature to be valid, the trusted QA Device (A) and ink QA Device (B) must share a *common* or a *variant* key i.e $B.K_{n1} = A.K_{n2}$ or $B.K_{n1} = FormKeyVariant(A.K_{n2}, B.ChipId)$.

Table 315. Command sequence for updating the ink-remaining (basic)

Seq No	Function	Parameter
1	B.WriteFields	VectNum = 0, FieldSelect =Select bits corresponding to the Ink fields, The ink field locations should have been determined before by using the method in Section `31.2.2.1 FieldVal= Decremented ink-remaining amount
		ResultFlag = Pass/Fail
2	A.Random	None
		R_A = RL
3	B.Read	KeyRef = n1, SigOnly = 1, (We only need the signature because we already know the data) MSelect = $_{M0}$, KeyIdSelect = 0, WordSelectForDesiredM = corresponds to the ink fields written in Seq No 1, RE= R_A
		If ResultFlag = Pass then SelectedWordsOfSelectedMs not returned because [SigOnly] = 1 in Seq 3, R_B = R_L, SIG_B = SIGout Refer to Section 15.3.1.
4	A.Test	KeyRef = n2, DataLength = length in words as per Seq No 1 [MVal] preformatted as per Section 16.1, Data = as per Seq No 1 [MVal] preformatted as per Section 16.1, RE =R_B, SIGE = SIG_B
		ResultFlag = Pass/Fail

31.4.3   Preauthorisation

5   This section describes the update of logical ink amounts using preauthorisation.

The basic preauthorisation sequence is as follows:

a.   Preauthorise *before* the first print. Preauthorisation amount depends on the printer model. Example amounts could be the ink required for an fully covered A4 page or an A3 page. Value corresponding to the preauth amount is written to the preauth field in the ink QA

10   Device.

Note: The preauth value must be correctly interpreted on different printer models i.e if a preauthorisation amount of A4 page is set in the ink cartridge in printer1(model1), and later the ink cartridge is placed in printer2(model2) with its preauth still set, printer2 must deduct an A4 page worth of ink from ink-remaining amount.

15   b.   Print the page.

c.   Write the deducted logical amount to the ink field of the ink QA Device and validate the write by reading the signature of the ink field.

d.   Repeat *b* to *c* till the last page has been printed.

e.   Clear the preauth amount.

f. If the power is cut off before the preauth is applied, on startup apply the preauth amount to the corresponding ink field, by performing a non authenticated write of the decremented amount and clear the preauth amount by performing an authenticated write of the preauth field.

5 *31.4.3.1 Set up of the preauth field*

Only a single preauth gield must exist in an Ink QA Device.

Preauth field will consist of a single $_{M0}$ word but can be optionally extended to two $_{M0}$ words by using a different value of *type* attribute. Figure 388 shows the setup of preauth field's attributes in $_{M1}$.

10 . The preauth field has authenticated ReadWrite access using the *INK_USAGE_KEY i.e INK_USAGE_KEY* can perform authenticated writes to this field. This key or its variant is shared between the ink QA Device and the printer QA Device to validate any data read from the ink cartridge. For signature to be valid, $B.K_{n1} = A.K_{n2}$ or $B.K_{n1} = FormKeyVariant(A.K_{n2}, B.ChipId)$, where $K_{n1} = INK\_USAGE\_KEY$. The system performs a *WriteAuth* to the preauth field using this

15 key, to set up the preauth amount, and to clear the preauth amount.

The preauth field is identified by two attributes:

• *Type* attribute - TYPE_PREAUTH . Refer to Appendix A.

• *KeyId* of *KeyNum* attribute must be the same as the *KeyId* of the *INK_USAGE_KEY* which the printer uses to validate the any data read from the ink QA

20 Device.

The Preauth field can be applied to a single ink field or multiple ink fields.

31.4.3.2 Preauth applied to a single ink field

In this case the entire preauth field is used to store the preauth amount and is only linked to one ink field.

25 *31.4.3.3 Preauth applied to multiple ink fields*

Multiple preauth fields can be accommodated in a single $_{M0}$ field by a scheme shown in Figure 388A.

This scheme supports a maximum of 8 ink fields being present in the Ink QA Device.

The field in $_{M0}$ is divided into two parts- preauth field select and preauth amount. Each bit in preauth

30 field select corresponds to a single ink field, and the preauth amount for each ink field is the same. If an ink cartridge uses multiple inks which are preauthorised, then each of the inks will have a corresponding preauth field bit. Before a particular ink is used for printing the corresponding preauth field bit is set. The preauth amount field is also set if the previous amount is zero. At finish, the preauth field bit is cleared. If more than one ink is used, the preauth bit for each ink field is set, and

35 at finish each bit is cleared with last bit clearing the preauth amount as well.

31.4.3.4 Locating preauth fields and determining preauth field value

893

The preauth field can be located in the same manner as the ink field. If the printer wants to find the preauth field in the ink QA Device, it must call the function *FindFieldNum* (see Section 31.2.2.2) with *keyIdRequired* = KeyId of Network_OEM_Ink_Usage_Key and *typeRequired* = TYPE_PREAUTH.

The preauth field value can be read in the same manner as the ink-remaining amount. This requires
5   using of the function *GetFieldDataWords(FieldNum,FieldData[], M0,M1)* described in Section 27.1.4.14. *FieldNum* must be set to the value returned from function *FindFieldNum*, which in this case is the field number of the preauth field. FieldData returns the value of the preauth field.

31.4.3.5 Command sequence

The command sequence can be broken up into three parts:
10   •   Start of print sequence.

•   During print sequence.

•   End of print sequence.

31.4.3.5.1   Start of print sequence

This sets up the preauth amount before the start of printing.
15   Table 316 shows the command sequence for start of print sequence. The first *Random- Read-Test* sequence determines the preauth field in the ink QA Device and its value. The *Random-SignM-WriteFieldsAuth* sequence, then writes to the preauth field the new preauth value.

Table 316. Updating the consumable remaining (preauth) start of print sequence

Seq No	Function	Parameters
Random-Read -Test sequence to determine the location of the preauth field in the ink QA Device and its value		
1	A.Random	None
		$R_A = RL$
2	B.Read	KeyRef = n1, SigOnly = 0, WordSelectForDesiredM (for $_{M0}$) = all 16 words of M0 and all 16 words of M1 MSelect = 0x03(indicates M0 and M1), KeyIdSelect = 0xFF (Read all KeyIds), WordSelectForDesiredM (for $_{M0}$)= 0xFFFF (Read all 16 $_{M0}$words), WordSelectForDesiredM (for $_{M1}$)= 0xFFFF(Read all 16 $_{M1}$words), RE= R_A
		If ResultFlag = Pass then MWords = SelectedWordsOfSelectedMs as per input [MSelect] and [WordSelectForDesiredM], $R_B = R_L$, SIG_B = SIGout Refer to Section 15.3.1
3	A.Test	KeyRef = n2, DataLength = length of MWords in words preformatted as per Section 16.1, Data = MWords as per Seq No 2 preformatted as per Section 16.1, RE =R_B, SIGE = SIG_B

		ResultFlag = Pass/Fail

Random-SignM-WriteFieldsAuth sequence to write the new preauth value		
4	B.Random	None
		R_{B1} = RL
5	A.SignM	KeyRef = n2, FieldSelect = Select bit corresponding to the Preauth field, FieldVal = new preauth value, ChipId = ChipId of B, R_E= R_{B1}
		If ResultFlag = Pass then R_{A1} = R_L SIG_A= SIGout Refer to Section 27.1.3.1
6	B.WriteFieldsAuth	KeyRef = n1, FieldSelect= same as Seq 5 [FieldSelect], FieldVal= same as Seq 5 [FieldVal], RE= R_{A1}, SIGE = SIG_A
		ResultFlag = Pass /Fail

31.4.3.5.2

During print sequence

5 This set of commands are repeated at equal intervals to update logical ink amounts to the ink QA Device during printing.

Table 317 shows the command sequence for the print sequence. The *WriteFields* writes the updated value to the ink field. *Random-Read-Test* reads back the value written and tests whether the value read matches the value written.

10 Table 317. Updating the consumable remaining (preauth) during print sequence

Seq No	Function	Parameters
Write the decremented ink-remaining account.		
7	B.WriteFields	FieldSelect = Select bits corresponding to the Ink fields, FieldVal= Decremented ink-remaining amount for a single ink or multiple ink fields as per FieldSelect.
		ResultFlag = Pass /Fail
Random-Read-Test sequence to read and verify the ink-remaining amount written		
8	A.Random	None
		R_A = RL
9	B.Read	KeyRef = n1, SigOnly = 1 -(We only need the signature because we already know the data), MSelect =0x01 (only $_{M0}$), KeyIdSelect = 0, WordSelectForDesiredM = corresponds to the ink fields written in Seq No 7, RE= R_A

		If ResultFlag = Pass then SelectedWordsOfSelectedMs not returned because [SigOnly] = 1 in Seq 9 $R_B = R_L$, $SIG_B = SIGout$ Refer to Section 15.3.1.
10	A. Test	KeyRef = n2, DataLength = length in words as per Seq No 7 [MVal] preformatted as per Section 16.1, Data = as per Seq No 7 [MVal] preformatted as per Section 16.1, RE =R_B, SIGE = SIG_B
		ResultFlag = Pass/Fail

31.4.3.5.3   End of print sequence

This sequence clears preauth amount before the print sequence is completed.

Table 318 shows the command sequence for the end of print sequence.

5   The preauth field is read using the *Random-Read-Test* sequence. And the preauth field is cleared using the *Random-SignM-WriteFieldsAuth* sequence.

Table 318. Updating the consumable remaining (preauth) end of print sequence

Seq No	Function	Parameters
Random-Read-Test sequence to read the preauth field and verify the preauth data		
11	A.Random	None
		$R_A = R_L$
12	B.Read	KeyRef = n1, SigOnly = 1, MSelect = 0x01(only M0), KeyIdSelect = 0, WordSelectForDesiredM (for $_{M0}$)= Words corresponding to the Preauthfield that has been written to in Seq 5 [FieldSelect] in Table 317. RE= R_A
		If ResultFlag = Pass then MWords = SelectedWordsOfSelectedMs as per Seq No 12 [MSelect] and [WordSelectForDesiredM], $R_B = R_L$, $SIG_B = SIGout$ Refer to Section 15.3.1
13	A.Test	KeyRef = n2, DataLength = length of MWords in words as per Seq No 12 preformatted as per Section 16.1, Data = MWords as per Seq No 12 preformatted as per Section 16.1, RE =R_B, SIGE = SIG_B
		ResultFlag = Pass/Fail
Random-SignM-WriteFieldsAuth sequence clears the preauth field		
14	B.Random	None
		$R_{B1} = R_L$
15	A.SignM	KeyRef = n2, FieldSelect =Select bit corresponding to Pre authfield, FieldVal = Clear the preauth field, ChipId = ChipId of B, R_E= R_{B1}
		If ResultFlag = Pass then R_{A1}= R_L SIG_A =SIGout Refer to Section 27.1.3.1

16	B.WriteFieldsAuth	KeyRef = n1, FieldNum = same as Seq 5 [FieldSelect], FieldData = same as Seq 5 [FieldVal], RE= R_{B1}, SIGE = SIG_A
		ResultFlag = Pass /Fail

31.4.4 Preauthorisation through the Translate function

This is performed when the system trusted QA Device doesn't share a key with the ink QA Device, and uses a translating QA Device to *Translate* a *Read* from the ink QA Device, and to *Translate* a
SignM to the ink QA Device.

The basic translate principle involves translating the *Read* data from the untrusted QA Device, to the *Test* data of the trusted QA Device, and translating the *SignM* data from the trusted QA Device, to the *WriteFieldsAuth* data of the untrusted QA Device.

For validating signatures using translation:

- The trusted QA Device (A) and the translating QA Device (C) must share a *common* or a *variant* key i.e C.K_{n3} = A.K_{n2} or C.K_{n3} = FormKeyVariant(A.K_{n2}, C.ChipId).

- The ink QA Device (B) and the translating QA Device (C) must share a *common* or a *variant* keyi.e C.K_{n2} = B.K_{n1} or B.K_{n1} = FormKeyVariant(C.K_{n2}, B.ChipId).

Only the start of print sequence is described using *Translate*. The rest of the sequences in preauthorisation can be modified to apply translation using this example.

Table 319 shows the command sequence for preauth (start of print sequence) using translation.

Table 319. Preauth(start of print sequence) using translate command

Seq No	Function	Parameter
*Random-Read-Random-Translate-Test sequence reads the location of the preauth field and its value using the translating QA Device C*		
1	C.Random	None
		R_C= RL
2	B.Read	KeyRef = n1, SigOnly = 0, MSelect = 0x03(indicates M0 and M1), KeyIdSelect = 0xFF (Read all KeyIds), WordSelectForDesiredM (for $_{M0}$)= 0xFFFF (Read all 16 $_{M0}$ words), WordSelectForDesiredM (for $_{M1}$)= 0xFFFF(Read all 16 $_{M1}$words), RE= R_A
		If ResultFlag = Pass then MWords = SelectedWordsOfSelectedMs as per input [MSelect] and [WordSelectForDesiredM], R_B = R_L, SIG_B = SIGout Refer to Section 15.3.1
3	A.Random	None
		R_A= RL

4	C.Translate	InputKeyRef =n2, DataLength (in words) = length of MWords in words as per Seq No 2 preformatted as per Section n 17.1, Data = MWords as returned from Seq No 2 preformatted as per Section 17.1, RE= R_B, SIGE = SIG_B OutputKeyRef = n3, RE2 = R_A
		If ResultFlag = Pass then R_{C1}= $RL2$, SIG_C = SIGOut Refer to Figure 15.3.1
5	A.Test	KeyRef = n2, DataLength = length of MWords in words as per Seq No 2 preformatted as per Section 16.1, Data = MWords as returned from Seq No 2 parameter preformatted as per Section 16.1, RE =R_{C1}, SIGE = SIG_C
		ResultFlag = Pass/Fail

Random-SignM-Random-Translate-WriteFieldAuth sequence to write the new preauth value using the translating QA Device C

6	C.Random	None
		$R_{C2=}$ R_L
7	A.SignM	KeyRef = n2, FieldSelect =Select bit corresponding to Pre authfield, FieldVal = new value of preauth field, ChipId = ChipId of B, R_E= R_{C2}
		If ResultFlag = Pass then R_{A1} = R_L SIG_A= SIGout Refer to Section 27.1.3.1
8	B.Random	None
		R_{B1} = R_L
9	C.Translate	InputKeyRef =n3, DataLength (in words) = length in words as per Seq 7 [FieldSelect] preformatted as per Section 17.1, Data = same as Seq 7 [FieldVal] preformatted as per Section 17.1, RE= R_{A1}, SIGE = SIG_A, OutputKeyRef = n2, RE2 = R_{B1}
		If ResultFlag = Pass then R_{C3}= R_{L2} SIG_C = SIGOut Refer to Figure 15.3.1
10	B.WriteFieldsAuth	KeyRef = n1, FieldNum = same as Seq 7 [FieldSelect], FieldData = same as Seq 7 [FieldVal], RE= R_{C3}, SIGE = SIG_C
		ResultFlag = Pass /Fail,

31.5   UPGRADING THE PRINTER PARAMETERS

This sequence is performed when a printer's operating parameter is upgraded.

The *Parameter Upgrader QA Device* stores the *upgrade value* which is copied to the operating
parameter field of the *Printer QA Device,* and the *count-remaining* associated with *upgrade value* is
decremented *by 1* in the *Parameter Upgrader QA Device.*

The Parameter Upgrader QA Device output the data and signature only after completing all
necessary checks for the upgrade.

31.5.1 Basic

The basic upgrade is used when the Parameter Upgrader QA Device and Printer QA Device being upgraded share a common key or a variant key i.e $B.K_{n1} = A.K_{n2}$ or $B.K_{n1} = FormKeyVariant(A.K_{n2}, B.ChipId)$, where B is the Printer QA Device and A is the Parameter Upgrader QA Device.

5   Therefore, the messages and their signatures, generated by each of them can be correctly interpreted by the other.

The transfer sequence is performed using *Random-Read-Random-XferField-WriteFieldsAuth* .

Table 320 shows the command sequence for a basic upgrade.

<div align="center">Table 320. Basic upgrade command sequence</div>

10

Seq No	Function	Parameter
Random-Read-Random-XferField-WriteFieldsAuth reads M0 and M1 of the QA Device being upgraded, Parameter Upgrader QA Device produces the upgrade value for FieldNumE and Sequence data fields SEQ_1 and SEQ_2, then these values are written to the Printer QA Device.		
1	A.*Random*	None
		$R_A = R_L$
2	B.*Read*	KeyRef = n1, SigOnly = 0, MSelect = 3 (indicates M_0 and M_1), KeyIdSelect = 0x00 (no KeyIds required), WordSelectForDesiredM (for M_0)= 0xFFFF (Read all M_0words), WordSelectForDesiredM (for M_1)= 0xFFFF(Read all M_1words), RE= R_A
		If ResultFlag = Pass then MWords = SelectedWordsOfSelectedMs, as per input [MSelect] and [WordSelectForDesiredM], R_B = RL,. SIG_B = SIGout Refer to Section 15.3.1
3	B.*Random*	None
		$R_{B1} = R_L$
4	A.*XferField*	KeyRef = n2, $_{M0}$OfExternal = First 16 words of MWords, $_{M1}$OfExternal= Last 16 words of MWords, ChipId = ChipId of B, FieldNumL= The field storing the upgrade value in the Parameter Upgrader QA Device. The value of this field will be copied to FieldNumE. FieldNumE= The field which will be upgraded in the Printer QA Device. $R_E= R_B$, $R_{E2} = R_{B1}$, SIG_E= SIG_B
		If ResultFlag = Pass then FieldSelectB1 = FieldSelect - Select bits for FieldNumE and Seq data fields SEQ_1 and SEQ_2 field, FieldValB1 = FieldVal -New Value for FieldNumE (Copied from FieldNumL of the Parameter Upgrader QA Device) and sequence data fields $R_{A1}= R_{L2}$, SIG_A = SIGout = Refer to Section 27.1.3.1.

| 5 | B.WriteFieldsAuth | KeyRef = n1, FieldSelect= FieldSelectB1, FieldData = FieldValB1, RE = R_{A1}, SIGE = SIG_A |
| | | ResultFlag = Pass/Fail |

31.5.2 Using the Translate function

The upgrade through the *Translate* function is used when the Parameter Upgrader QA Device and the Printer QA Device don't share a key between them. The translating QA Device shares a key

5 with the Parameter Upgrader QA Device and a second key with the Printer QA Device. Therefore the messages and their signatures, generated by the Parameter Upgrader QA Device and the Printer QA Device are translated appropriately by the translating QA Device. The translating QA Device validates the *Read* from the Printer QA Device, and translates it for input to the *XferField* function. The translating QA Device will validate the output from the *XferField* function, and then

10 translate it for input to *WriteFieldsAuth* message of the Printer QA Device.

For validating signatures using translation:

- The Parameter Upgrader QA Device (A) and the translating QA Device (C) must share a *common* or a *variant* key i.e $C.K_{n3}$ = $A.K_{n2}$ or $C.K_{n3}$ = FormKeyVariant($A.K_{n2}$, C.ChipId).

- The Printer QA Device (B) and the translating QA Device (C) must share a *common* or a

15 *variant* key i.e $C.K_{n2}$ = $B.K_{n1}$ or $B.K_{n1}$ = FormKeyVariant($C.K_{n2}$, B.ChipId).

Table 321 shows the command sequence for a basic refill using translation.

Table 321. An upgrade with translate command sequence

Seq No	Function	Command

Seq No	Function	Command
Random-Read-Random-Translate-Random-XferField-Random-Translate-Random-WriteFieldsAuth reads M0 and M1of the Printer QA Device using the translating QA Device C and then does a write of the upgrade value to FieldNumE and new sequence data to the seq data fields SEQ_1 and SEQ_2 field of the Printer QA Device using the translating QA Device C.		
1	C.Random	None
		$R_C = R_L$
2	B.Read	KeyRef = n1, SigOnly = 0, MSelect =0x03(indicates $_{M0}$ and $_{M1}$), KeyIdSelect = 0x00 (no KeyIds required), WordSelectForDesiredM (for $_{M0}$)= 0xFFFF (Read all $_{M0}$words), WordSelectForDesiredM (for $_{M1}$)= 0xFFFF(Read all $_{M1}$ words), $R_E = R_C$
		If ResultFlag = Pass then MWords = SelectedWordsOfSelectedMs as per input [MSelect] and [WordSelectForDesiredM], R_B = RL, SIG_B = SIGout Refer to Section 15.3.1
3	A.Random	None

900

		$R_A = R_L$
4	C.*Translate*	InputKeyRef =n2, DataLength = MWords length in words as per Seq No 2 preformatted as per Section 17.1, Data = MWords as returned from Seq No 2 preformatted as per Section 17.1, RE= R_B, SIGE= SIG_B, OutputKeyRef = n3, RE2 = R_A
		If ResultFlag = Pass then R_{C1} = RL2, SIG_C= SIGOut Refer to Section 17.3.1
5	C.*Random*	None
		R_{C2}= R_L
6	A.*XferField*	KeyRef = n2, $_{M0}$OfExternal = First 16 words of MWords, $_{M1}$OfExternal= Last 16 words of MWords, ChipId = ChipId of B, FieldNumL= The field storing the upgrade value in the Parameter Upgrader QA Device. FieldNumE= The field which will be upgraded in the Printer QA Device. R_E= R_{C1}, R_{E2} = R_{C2}, SIG_E= SIG_C
		If ResultFlag = Pass then FieldSelectB1 = FieldSelect - Select bits for FieldNumE and sequence fields, FieldValB1 = FieldVal -New Value for FieldNumE (Copied from FieldNumL of the Parameter Upgrader QA Device) and sequence fields SEQ_1 and SEQ_2, R_{A1}= R_{L2}, SIG_A = SIGout Refer to Section 27.1.3.1
7	B.*Random*	None
		R_{B1}= R_L
8	C.*Translate*	InputKeyRef =n3, DataLength = FieldValB1 length in words as per Seq No 6 preformatted as per Section 17.1, Data = FieldValB1 as returned from Seq No 6 preformatted as per Section 17.1, RE= R_{A1}, SIGE = SIG_A, OutputKeyRef= n2, RE2 = R_{B1}
		If ResultFlag = Pass then R_{C3} = R_{L2}, SIG_C= SIGOut Refer to Section 17.3.1
19	B.*WriteFieldsA uth*	KeyRef = n1, FieldSelect = FieldSelectB1, FieldVal = FieldValB1, RE = R_{C3}, SIGE = SIG_C
10		ResultFlag = Pass/Fail

31.6 RECOVERING FROM A FAILED UPGRADE

This sequence is performed if the upgrade failed (for e.g Printer QA Device didn't receive the upgrade message correctly and hence didn't upgrade successfully). The Parameter Upgrader QA Device therefore needs to be rolled back to the previous value before the upgrade. In this case, the count-remaining associated with the upgrade value in the Parameter Upgrader QA Device is increased by one.

The Parameter Upgrader QA Device checks that the Printer QA Device didn't actually receive the message correctly using the *StartRollBack* function. The *RollBackField* performs further comparisons on *sequence* fields and *FieldNumE* of the Printer QA Device to values stored in the XferEntry cache. After performing all checks, the Parameter Upgrader QA Device increments the

count remaining field associated with the *upgrade value* field by one. Refer to Section 26 and Section 28 for details.

The rollback is started using the *Random-Read-Random-StartRollBack-WriteFieldsAuth* and the rollback of the Parameter Upgrader QA Device is performed using *Random-Read-RollBackField*

5   sequence.

Table 322 shows the command sequence for a rollback upgrade.

Seq No	Function	Command
Random-Read-Random-StartRollBack-WriteFieldsAuth starts the rollback and updates data for the sequence fields.		
1	A.*Random*	None
		$R_A = RL$
2	B.*Read*	KeyRef = n1, SigOnly = 0, MSelect =0x03(indicates $_{M0}$ and $_{M1}$), KeyIdSelect = 0x00 (no KeyIds required), WordSelectForDesiredM (for $_{M0}$)= 0xFFFF (Read all $_{M0}$words), WordSelectForDesiredM (for $_{M1}$)= 0xFFFF(Read all $_{M1}$words), $R_E = R_A$
		If ResultFlag = Pass then MWords = SelectedWordsOfSelectedMs as per input [MSelect] and [WordSelectForDesiredM], $R_B = R_L$, SIG_B = SIGout Refer to Section 15.3.1
3	B.*Random*	None
		$R_{B1} = R_L$
4	A.*StartRoll Back*	KeyRef = n2, $_{M0}$OfExternal = First 16 words of MWords, $_{M1}$OfExternal= Last 16 words of MWords, ChipId = ChipId of B, FieldNumE= The field which was not upgraded in the Printer QA Device, FieldNumL = The upgrade value in the Parameter Upgrader QA Device which couldn't be copied to FieldNumE of the Printer QA Device, $R_E = R_B$, $R_{E2} = R_{B1}$, $SIG_E = SIG_B$
		If ResultFlag = Pass then FieldSelectB = FieldSelect - Select bits for sequence data fields SEQ_1 and SEQ_2, FieldValB = FieldVal - New values for SEQ_1 and SEEQ_2 fields $R_{A1} = R_{L2}$ SIG_A = SIGout Refer to Section 27.1.3.1.
5	B.*WriteFiel dsAuth*	KeyRef = n1, FieldSelect= FieldSelectB, FieldData = FieldValB, RE = R_{A1}, SIGE = SIG_A
		ResultFlag = Pass/Fail
Random-Read-RollBackField performs a read of the QA Device being upgraded, checks its values are as per Xfer Entry cache, and then adjusts its count-remaining field.		
6	A.*Random*	None
		$R_{A2} = RL$

902

7	B.Read	KeyRef = n1, SigOnly = 0, MSelect =0x03(indicates $_{M0}$ and $_{M1}$), KeyIdSelect = 0x00 (no KeyIds required), WordSelectForDesiredM (for $_{M0}$)= 0xFFFF (Read all $_{M0}$words), WordSelectForDesiredM (for $_{M1}$)= 0xFFFF(Read all $_{M1}$words), R_E= R_{A2}
		If ResultFlag = Pass then MWords = SelectedWordsOfSelectedMs as per input [MSelect] and [WordSelectForDesiredM], R_{B2} = RL, SIG_B = SIGout Refer to Section 15.3.1
8	A.RollBack Field	KeyRef = n2, $_{M0}$OfExternal = First 16 words of MWords, $_{M1}$OfExternal= Last 16 words of MWords, ChipId = ChipId of B, FieldNumE= The field which was not upgraded in the Printer QA Device, FieldNumL = The upgrade value in the Parameter Upgrader QA Device which couldn't be copied to FieldNumE of the Printer QA Device, R_E= R_{B2}, SIG_E= SIG_B
		ResultFlag = Pass/Fail

31.7 RE/FILLING THE CONSUMABLE (INK)

This sequence is performed when an ink cartridge is first manufactured or after all the physical ink has been used, it can be filled or refilled. The re/fill protocol is used to transfer the logical ink from
5   the Ink Refill QA Device to the Ink QA Device in the ink cartridge.

The Ink Refill QA Device stores the amount of logical ink corresponding to the physical ink in the refill station. During the refill, the required logical amount (corresponding to the physical transfer amount) is transferred from the Ink Refill QA Device to the Ink QA Device.

The Ink Refill QA Device output the transfer data only after completing all necessary checks to
10   ensure that correct logical ink type is being transferred e.g Network_OEM1_infrared ink is not transferred to Network_OEM2_cyan ink. Refer to the *XferAmount* command in Section 27.1.

31.7.1 Basic refill

The basic refill is used when the Ink Refill QA Device and the Ink QA Device share a common key or a variant key i.e $B.K_{n1}$ = $A.K_{n2}$ or $B.K_{n1}$ = FormKeyVariant($A.K_{n2}$, B.ChipId) where B is the Ink QA
15   Device and A is the Ink Refill QA Device. Therefore, the messages and their signatures, generated by each of them can be correctly interpreted by the other.

The *Xfer Sequence* is started using *Random-Read-Random-StartXfer-WriteAuth* and the the *Xfer Amount* is written to the QA Device being refilled using *Random-Read-Random-XferAmount-WriteFieldsAuth* sequence.

20

Table 323 - the command sequence for a basic refill.

Seq No	Function	Parameter
		Random-Read-Random-XferAmount-WriteFieldsAuth reads M0 and M1 of the Ink QA Device being refilled, produce updated amount for FieldNumE and sequence datat field by calling XferAmount on Ink Refill QA Device, and finally writing the updated value to Ink QA Device using WriteFieldsAuth.*
1	*A.Random*	None
		$R_A = R_L$
2	*B.Read*	KeyRef = n1, SigOnly = 0, MSelect = 0x03(indicates $_{M0}$ and $_{M1}$), KeyIdSelect = 0x00 (no KeyIds required), WordSelectForDesiredM (for $_{M0}$)= 0xFFFF (Read all $_{M0}$words), WordSelectForDesiredM (for $_{M1}$)= 0xFFFF(Read all $_{M1}$words), RE= R_A
		If ResultFlag = Pass then MWords = SelectedWordsOfSelectedMs as per input [MSelect] and [WordSelectForDesiredM], R_B = RL, SIG_B = SIGout Refer to Section 15.3.1
3	*B.Random*	None
		$R_{B1}= R_L$
4	*AxferAmount*	KeyRef = n2, $_{M0}$OfExternal = First 16 words of MWords, $_{M1}$OfExternal= Last 16 words of MWords, ChipId = ChipId of B, FieldNumL= ink-remaining field of the Ink Refill QA Device, FieldNumE= ink-remaining field of the Ink QA Device, XferValLength = length in words of XferVal XferVal = Value to be transferred from Ink Refill QA Device to Ink QA Device being refilled, $R_{E=} R_B$, $R_{E2 =} R_{B1}$, $SIG_E= SIG_B$
		If ResultFlag = Pass then FieldSelectB1 = FieldSelect - Select bits for FieldNumE and sequence data field SEQ_1 and SEQ_2, FieldValB1 = FieldVal -New Value for FieldNumE (transferred from FieldNumL of the Ink Refill QA Device) and sequence data fields SEQ_1 and SEQ_2, R_{A1} = R_{L2}, SIG_A = SIGout Refer to Section 27.1.3.1.
5	*B.WriteFieldsAuth*	KeyRef = n1, FieldSelect= FieldSelectB, FieldData = FieldValB, RE = R_{A1}, SIGE = SIG_A
		ResultFlag = Pass/Fail

5   31.7.2   Using the Translate function

The refill through the *Translate* function is used when the Ink Refill QA Device and the Ink QA Device don't share a key between them. The translating QA Device shares a key with the Ink Refill

QA Device and a second key with the Ink QA Device. Therefore the messages and their signatures, generated by the Ink Refill QA Device and the Ink QA Device, are translated appropriately by the translating QA Device. The translating QA Device validates the *Read* from the Ink QA Device, and translates it for input to the *XferAmount* function. The translating QA Device will validate the output

5   from the *XferAmount* function, and then translate it for input to *WriteFieldsAuth* message of the Ink QA Device.

For validating signatures using translation:

- The Ink Refill QA Device (A) and the translating QA Device (C) must share a *common* or a *variant* key i.e $C.K_{n3} = A.K_{n2}$ or $C.K_{n3} = FormKeyVariant(A.K_{n2}, C.ChipId)$.

10   - The Ink Refill QA Device being refilled (B) and the translating QA Device (C) must share a *common* or a *variant* key i.e $C.K_{n2} = B.K_{n1}$ or $B.K_{n1} = FormKeyVariant(C.K_{n2}, B.ChipId)$.

Table 324. A basic refill using translation command sequence

Seq No	Function	Command
Random-Read-Random-Translate-Random-XferAmount-Random-Translate-Random-WriteFieldsAuth - reads M0 and M1 of the Ink QA Device being refilled using the translating QA Device C , produce updated amount for FieldNumE and sequence data field by calling XferAmount on Ink Refill QA Device, and finally writing the updated value to Ink QA Device using the translating QA Device.		
1	C.Random	None
		$R_C = R_L$
2	B.Read	KeyRef = n1, SigOnly = 0, MSelect =0x03(indicates $_{M0}$ and $_{M1}$), KeyIdSelect = 0x00 (no KeyIds required), WordSelectForDesiredM (for $_{M0}$)= 0xFFFF (Read all $_{M0}$words), WordSelectForDesiredM (for $_{M1}$)= 0xFFFF(Read all $_{M1}$words), $R_E = R_C$
		If ResultFlag = Pass then MWords = SelectedWordsOfSelectedMs as per input [MSelect] and [WordSelectForDesiredM], $R_B = R_L$, SIG_B = SIGout Refer to Section 15.3.1
3	A.Random	None
		$R_A = R_L$
4	C.Translate	InputKeyRef =n2, DataLength = MWords length in words as per Seq No 2 preformatted as per Section 17.1, Data = MWords as returned from Seq No 2 preformatted as per Section 17.1, RE= R_B, SIGE= SIG_B, OutputKeyRef = n3, RE2 = R_A
		If ResultFlag = Pass then $R_{C1} = R_{L2}$, SIG_C= SIGOut Refer to Section 17.3.1
5	C.Random	None
		$R_L = R_{C2}$

905

6	A.XferAmount	KeyRef = n2, $_{M0}$OfExternal = First 16 words of MWords, $_{M1}$OfExternal= Last 16 words of MWords, ChipId = ChipId of B, FieldNumL= ink-remaining field of the Ink Refill QA Device, FieldNumE= ink-remaining field of the Ink QA Device, XferValLength = length in words of XferVal XferVal = Value to be transferred from Ink Refill QA Device to Ink QA Device being refilled, R_E= R_{C1}, R_{E2} = R_{C2}, SIG_E= SIG_C
		If ResultFlag = Pass then FieldSelectB1 = FieldSelect - Select bits for FieldNumE and sequence data field SEQ_1 and SEQ_2, FieldValB1 = FieldVal -New Value for FieldNumE (transferred from FieldNumL of the Ink Refill QA Device) and sequence data fields SEQ_1 and SEQ_2, R_{A1}= R_{L2}, SIG_A = SIGout Refer to Section 27.1.3.1
7	B.Random	None
		R_{B1} = R_L
8	C.Translate	InputKeyRef =n3, DataLength = FieldValB length in words as per Seq No 6 preformatted as per Section 17.1, Data = FieldValB as returned from Seq No 6 preformatted as per Section 17.1, RE= R_{A1}, SIGE = SIG_A OutputKeyRef= n2, RE2 = R_{B1}
		If ResultFlag = Pass then R_{C3} = RL2, SIG_C= SIGOut Refer to Section 17.3.1
9	B.WriteFieldsAuth	KeyRef = n1, FieldSelect= FieldSelectB, FieldData = FieldValB, RE = R_{C3}, SIGE = SIG_C
		ResultFlag = Pass/Fail

31.8 RECOVERING FROM A FAILED REFILL

This sequence is performed if the refill failed (for e.g Ink QA Device didn't receive the refill message correctly and hence didn't refill successfully). The Ink Refill QA Device therefore needs to be rolled back to the previous value before the refill.

The Ink Refill QA Device checks that the Ink QA Device didn't actually receive the message correctly using the *StartRollBack* function. The *RollBackAmount* performs further comparisons on *sequence* data field and *FieldNumE* of the Ink QA Device, to values stored in the *XferEntry* cache. After performing all checks, the Ink Refill QA Device adjusts its ink field to a previous value before the transfer request was processed by it. Refer to Section 26 and Section 28 for details.

The rollback is started using the *Random-Read-Random-StartRollBack-WriteFieldsAuth* and the rollback of the Ink Refill QA Device is performed using *Random-Read-RollBackAmount* sequence.

Table 325. Rollback amount command sequence

Seq No	Function	Command
over all	Random-Read-Random-StartRollBack-WriteAuth starts the rollback and updates data for the sequence data fields SEQ_1 and SEQ_2.	
1	A.Random	None
		R_A = RL
2	B.Read	KeyRef = n1, SigOnly = 0, MSelect =0x03(indicates $_{M0}$ and $_{M1}$), KeyIdSelect = 0x00 (no KeyIds required), WordSelectForDesiredM (for $_{M0}$)= 0xFFFF (Read all $_{M0}$words), WordSelectForDesiredM (for $_{M1}$)= 0xFFFF(Read all $_{M1}$words), R_E= R_A
		If ResultFlag = Pass then MWords = SelectedWordsOfSelectedMs as per input [MSelect] and [WordSelectForDesiredM], R_B = RL, SIG_B = SIGout Refer to Section 15.3.1
3	B.Random	None
		R_{B1}= R_L
4	A.StartRollBack	KeyRef = n2, $_{M0}$OfExternal = First 16 words of MWords, $_{M1}$OfExternal= Last 16 words of MWords, ChipId = ChipId of B, FieldNumL= ink-remaining field of the Ink Refill QA Device which will be adjusted to the value before the failed refill, FieldNumE= ink-remaining field of the Ink QA Device which failed to refill, R_E= R_B, R_{E2} = R_{B1} SIG_E= SIG_B
		If ResultFlag = Pass then FieldSelectB = FieldSelect - Select bits for sequence data fields- SEQ_1 and SEQ_2, FieldValB = FieldVal - New value for sequence data fields SEQ_1 and SEQ_2 R_{A1} = R_{L2}, SIG_A = SIGout Refer to Section 27.1.3.1.
5	B.WriteFieldsAuth	KeyRef = n1, FieldSelect= FieldSelectB in Seq No 4, FieldData = FieldValB in Seq No 4 RE = R_{A1}, SIGE = SIG_A
10		ResultFlag = Pass/Fail
over all	Random-Read-RollBackAmount performs a read of the Ink QA Device, checks its values are as per Xfer Entry cache, and then adjusts its ink-remaining field.	
11	A.Random	None
		R_{A2}. = RL
12	B.Read	KeyRef = n1, SigOnly = 0, MSelect =0x03(indicates $_{M0}$ and $_{M1}$), KeyIdReq = 0 (not required), KeyIdSelect = 0x00 (no KeyIds required), WordSelectForDesiredM (for $_{M0}$)= 0xFFFF (Read all $_{M0}$words), WordSelectForDesiredM (for $_{M1}$)= 0xFFFF(Read all

		$_{MI}$words), RE= R_{A2}
		If ResultFlag = Pass then MWords = SelectedWordsOfSelectedMs as per input [MSelect] and [WordSelectForDesiredM], R_{B2} = R_L, SIG_B = SIGout Refer to Section 15.3.1
13	A.RollBackAmount	KeyRef = n2, $_{M0}$OfExternal = First 16 words of MWords, $_{MI}$OfExternal= Last 16 words of MWords, ChipId = ChipId of B, FieldNumL= ink-remaining field of Ink Refill QA Device which will be adjusted to the value before the failed refill, FieldNumE= ink-remaining field of Ink QA Device which failed to refill, R_E= R_{B2}, SIG_E= SIG_B
		ResultFlag = Pass/Fail

31.9 UPGRADING/REFILLING/FILLING THE UPGRADER

This sequence is performed when a *count-remaining* field in the Parameter QA Device must be updated or when the *ink-remaining* field in the Ink Refill QA Device requires re/filling.

5   In case of the Parameter QA Device, another Parameter Upgrader Refill QA Device transfers its *count-remaining* value to the Parameter QA Device using the *transfer sequence* described in Section 31.4. Also refer to Section 28.6. This means the *count-remaining* in the Paramater Upgrader Refill QA Device must be decremented by the same amount that Parameter Upgrader QA Device is incremented by i.e a credit transfer occurs.

10   In case of the Ink Refill QA Device, another Ink Refill QA Device transfers its *ink-remaining* value to the Ink Refill QA Device using the *transfer sequence* described in Section 31.4. Also refer to Section 26.4. This means the logical *ink-remaining* in the *Ink* Refill QA Device must be decremented by the same amount that QA Device being refilled is incremented by i.e a credit transfer occurs.

32 Setting up for field use

15   This section consists of setting up the data structures in the QA Device correctly for field use. All data structures are first programmed to factory values. Some of the data structures can then be changed to application specific values at the ComCo or the OEM, while others are set to fixed values.

32.1 INSTANTIATING THE QA CHIP LOGICAL INTERFACE

20   This sequence is performed when the QA Device is first created. Table 326 shows the data structure on final program load.

Table 326. Data structure set up during final program load

Data Structure Name	Value Set to	Fixed or Updatable
ChipId	Unique Identifier for QA Device	Fixed
NumKey	Number of keys the QA Device can hold	Fixed
K_n	All $K_n = K_{batch}$. The K_{batch} is unique for a production batch[a].	Updateable if previous value is known
KeyId	All KeyIds = KeyId of K_{batch}.	Updateable along with K_n.
KeyLock	All KeyLock = unlocked	Updateable
NumVectors	Number of memory vectors in the QA Device.	Fixed
M0	Set to zeros	Updateable
M0	Set to zeros	Updateable
M_{2+}	Set to zeros	Updateable
P_n	Set to ones	Updateable
R	Set to an initial random value	Updateable

Each key slot has the same K_{batch}. If each key slot had a different K_{batch}, and any one of the K_{batch}

5   was compromised then the entire batch would be compromised till the K_{batch} was replaced to another key. Hence, each key slot having a different K_{batch} doesn't have any security advantages but requires more keys to be managed.

32.2   SETTING UP APPLICATION SPECIFIC DATA

The section defines the sequences for configuring the data structures in the QA Device to

10   application specific data.

32.2.1   Replacing keys

The QA Devices are programmed with production batch keys at final program load. The *COMCO* keys replace the production batch keys before the QA Devices are shipped to the ComCo. The ComCo replaces the *COMCO* keys to *COMCO_OEM* when shipping QA Devices to its OEMs.

15   The OEM replaces the *COMCO_OEM* to *COMCO_OEM_app* as the QA Devices are placed in ink cartridges or printers.

The replacement occurs without the ComCo or the OEM knowing the actual value of the key. The actual value of the keys is only to known to QACo. The ComCo or the OEM is able to perform these replacements because the QACo provides them with a key programming QA Device with keys

20   appropriately set which can generate the necessary messages and signatures to replace the old key with the new key.

Table 327 shows the command sequence for ReplaceKey. The *GetProgramKey* gets the new encrypted key from the key programming QA Device, and the encrypted new key is passed into the

QA Device whose key is being replaced through the *ReplaceKey* function. Depending on the *OldKeyRef* and *NewKeyRef* objects a *common encrypted key* or a *variant encrypted key* can be produced for the *ReplaceKey* function

Table 327. ReplaceKey command sequence

5

Seq No	Function	Command
1	*B.Random*	None
		$R_B = R_L$
2	*A.GetProgramKey*	OldKeyRef = Key Num of the old key. This key must be changed to the NewKeyRef in the QA Device whose key s being replaced. ChipId = Chip identifier of the QA Device whose key is being replaced. RE= R_B KeyLock = Set depending on whether the new key is the final key for the key slot or it will be replaced further. NewKeyRef = Key Num of the new key. This key will change the OldKeyRef in the QA Device whose key is being replaced.
		If ResultFlag = Pass then $R_A = RL$, KeyId$_{new}$ = KeyIdOfNewKey EncryptedNewKey = EncryptedKey SIGA = SIGout Refer to Section 22.2.1.
3	*B.ReplaceKey*	KeyNumToBeReplaced = Old key number, the old key could be a common key or a variant key, KeyId = KeyId$_{new}$, EncryptedKey= EncryptedNewKey, RE = RA, SIGE = SIGA
		ResultFlag = Pass/Fail

32.2.2 Setting up ReadOnly data

This sets the permanent functional parameters of the application where the QA Device has been placed. These parameters remain unchanged for the lifetime of the QA Device. In case of the ink

10 cartridge such parameters are colour and viscosity of the ink. These values are written to M_{2+} memory vectors using the *WriteM1+* function, and its permissions are set to ReadOnly by *SetPerm* function. These values are typically set at the OEM.

Table 328 shows the command sequence for setting up ReadOnly data.

Table 328. ReadOnly data setup command sequence

15

Seq No	Function	Command
1	*B.WriteM1+*	*VectNum* = 2 or 3, *WordSelect* = the selected words to be written, *MVal* = words corresponding to word select starting from LSW

		ResultFlag = Pass/Fail
2	B.SetPerm	(VectNum =same as Seq No 1parameter [VectNum], PermVal =same as Seq No 1 parameter [WordSelect])
		If ResultFlag = Pass then CurrPerm = NewPerm Current permission value after applying PermVal

In case of the SBR4320, the values written to M_{2+} memory vectors is write-once only i.e they are set to ReadOnly as soon as they are written to once, therefore the command sequence consists only of Seq No 1 in Table 329.

32.2.3   Defining fields in $_{M0}$

The QACo must determine the field definitions for M0 depending on the application of the QA Device. These field definitions will consist of the following:

- Number of fields and the size of each field.
- The Type attribute of each field.
- The access permission for each field.

Following fields have been presently defined in an ink QA Device:

- ink-remaining field. See Section 26 for details.
- Preauthorisation field. See Section 31.4.3 for details.
- Sequence data fields SEQ_1 and SEQ_2. See Section 26 for details.

Following fields have been presently defined in a printer QA Device:

- Operating parameter field.See Section 28 for details.
- Sequence data fields SEQ_1 and SEQ_2. See Section 26 for details.

After the field definitions are determined, they are formatted as per Section 8.1.1.4. These formatted values are then written to $_{M1}$ using a *WriteM1+* function.

Table 329. Defining M0 fields command sequence

Sequence No	Function	Command
1	B.WriteM1+	VectNum = 1, WordSelect = The selected words corresponding to the attribute field/fields of $_{M0}$, MVal = words corresponding to word select starting from LSW)
		ResultFlag = Pass/Fail

32.2.4   Writing values to fields in $_{M0}$

911

The writing of $_{M0}$ fields for an Ink QA Device will typically occur when the ink cartridge is filled with physical ink for the first time, and the equivalent logical ink is written to the Ink QA Device. Refer to Section 31.7 for details.

The writing of $_{M0}$ fields for a Printer QA Device will typically occur when the printer parameters are written for the first time. The procedure for writing of a printer parameter for the first time or upgrading a printer parameters is exactly the same. Refer to Section 31.5 for details.

Before any value is written to a field, the key slot containing the key which has authenticated ReadWrite access to the field must be locked.

Both Ink QA Device and Printer QA Device has a sequence data fields SEQ_1 and SEQ_2 as described in Section 27. These two fields must be initialised to 0xFFFFFFFF, refer to Section 27 for details.

The Ink QA Device/Printer QA Device and the trusted QA Device writing to it, share the *sequence* key or a variant *sequence* key between them i.e $B.K_{n1} = A.K_{n2}$ or $B.K_{n1}$ = FormKeyVariant($A.K_{n2}$, B.ChipId), where B is the Ink QA Device/Printer QA Device and A is the trusted QA Device. The command sequence used is described in Table 330.

Table 330. Command sequence for writing sequence data fields to the QA Devices.

Sequence No	Function	Parameters
1	*B.Random*	
		$R_B = RL$
2	*A.SignM*	KeyRef = n2, FieldSelect =Select bit correponding to SEQ_1 and SEQ-2 FieldVal = both fields set 0xFFFFFFFF. Refer to Section 31.4.3.3 ChipId = ChipId of B, $R_E = R_B$
		If ResultFlag = Pass then $R_A = R_L$ SIG_A =SIGout Refer to Section 27.1.3.1
3	*B.WriteFieldsAuth*	KeyRef = n1, FieldSelect = same as Seq 2[FieldSelect], FieldVal = same as Seq 2[FieldVal], RE= R_A, SIGE = SIG_A
		ResultFlag = Pass /Fail

32.3   SETTING UP THE UPGRADING QA DEVICE

The upgrading QA Device must be set up either as an Ink Refill QA Device or as a Parameter Upgrader QA Device.

Each upgrading QA Device must go through the following set up:

- The upgrading QA Device must be set to factory defaults. Refer to Section 32.1. At the end of this process the upgrading QA Device is either an Ink Refill QA Device or a Parameter Upgrader QA Device with production batch keys and M0 fields set to deafult.

912

- The upgrading QA Device must be programmed with the appropriate keys and upgrade data before it can start upgrading other QA Devices. Following must be performed on each upgrade QA Device:

a. The upgrading QA Device must be programmed with the appropriate keys required to upgrade other QA Devices and to upgrade itself when necessary.

b. The M0 fields must be correctly defined and set in M1.

For a Ink Refill QA Device the ink-remaining field must be defined and set. For a printer upgrade QA Device the *upgrade value* field and the *count-remaining* field must be defined and set.

All upgrade QA Devices must also have a sequence datat fields SEQ_1 and SEQ_2 which are used to upgrade the upgrading QA Device itself.

c. Finally, M0 fields defined in b must be written with appropriate values so that the upgrade QA Device can perform upgrades.

An Ink Refill QA Device will typically store the logical ink equivalent to the physical ink in a refill station, hence the Ink Refill QA Device's ink-remaining field must be written with the equivalent logical ink amount.

For a Parameter Upgrader QA Device the upgrade value field and the count-remaining field must be written. The upgrade value depends on the type of upgrade the Parameter Upgrader QA Device can perform i.e one Parameter Upgrader QA Device can upgrade to 10 ppm (pages per minute) while another Parameter Upgrader QA Device can upgrade to 5ppm. The *count-remaining* is the number of times the *Parameter Upgrader QA Device* is permitted to write the associated upgrade value to other QA Devices. The count-remaining field must be written to a positive non-zero value for the Parameter Upgrader QA Device to perform successful upgrades. Refer to Section 32.3.1 and Section 32.3.2 for details.

32.3.1   Setting up the Ink Refill QA Device

32.3.1.1 Setting up the keys

The Ink Refill QA DeviceQA Device could be transferring ink between peers or transferring ink down the heirachy, accordingly the peer to peer Ink Refill QA Device has *two keys (fill/refill* key and *sequence key*) as described in Section 27, and a Ink Refill QA Device transferring down the heirachy has *three keys (fill/refill* key, *transfer key and sequence key)*. These keys must be programmed into the Ink Refill QA Device using the sequence described in Section 32.2.1.

The Key Programming QA Device must be programmed with the appropriate production batch keys , and the fill/refill, transfer key and sequence key

The GetProgramKey function is called on the Key Programming QA Device with OldKeyRef (OldKeyRef - refer to Section 32.2.1) pointing to a production batch key, and the NewKeyRef (NewKeyRef - refer to Section 32.2.1) pointing to either a fill/refill key or a transfer key or a sequence key. The outputs from the GetProgramKey (signature and encrypted New Key) is passed in to ReplaceKey function of the Ink Refill QA Device.

913

The GetProgramKey function must be called (on the Key Programming QA Device) for replacing each of the production batch keys in the Ink Refill QA Device. The output of the GetProgramKey will be passed in to the ReplaceKey function called on the Ink Refill QA Device. The successful processing of the ReplaceKey function will replace an old key(production keys) to a corresponding

5 new key (either a fill/refill key or a transfer key or a sequence key).

32.3.1.2 Setting up the M0 field information in $_{M1}$

The *ink-remaining* field and the sequence data fields *SEQ_1 and SEQ_2* must be defined and set in the Ink Refill QA Device using the sequence described in Section 32.2.3.

32.3.1.3 Transferring ink amounts

10 Finally, the logical ink amounts are transferred to the ink-remaining field using the sequence described in Section 31.7.

The QACo will transfer to the ComCo Ink Refill QA Device at the top of the heirachy using the command sequence in Table 331.

For a successful transfer from QACo to ComCo, ComCo and QACo must share a *common* key or a

15 *variant* key be i.e ComCo.K_{n1} = QACo.K_{n2} or ComCo.K_{n1} = FormKeyVariant(QACo.K_{n2} ,ComCo.ChipId)K_{n1} is the *fill/refill* key for the ComCo refill QA Device..

Table 331. Command sequence for writing ink-remaining amounts to the highest QA Device in the heirachy.

Sequence No	Function	Parameters
1	B.Random	
		$R_B = RL$
2	A.SignM	KeyRef = n2, FieldSelect =Select bit correponding to the ink-remaining field, FieldVal = Ink amount to be transferred, Refer to Section 31.4.3.3 ChipId = ChipId of B, R_E= R_B
		If ResultFlag = Pass then R_A= R_L SIG_A =SIGout Refer to Section 27.1.3.1
3	B.WriteFieldsAuth	KeyRef = n1, FieldSelect = same as Seq 2[FieldSelect], FieldVal = same as Seq 2[FieldVal], RE= R_A, SIGE = SIG_A
		ResultFlag = Pass /Fail

20

32.3.1.4 Setting up sequence data fields

The Ink Refill QA Device has sequence data fields SEQ_1 and SEQ_2 (as described in Section 27) because its ink-remaining fields can be refilled as well. These two fields must be initialised to 0xFFFFFFFF, refer to Section 27 for details.

914

The Ink Refill QA Device and the trusted QA Device writing to it, share the *sequence* key or a variant *sequence* key between them i.e $B.K_{n1} = A.K_{n2}$ or $B.K_{n1} = FormKeyVariant(A.K_{n2}, B.ChipId)$, where B is the Ink Refill QA Device and A is the trusted QA Device. The command sequence used is described in Table 331.

5   32.3.2   Setting up the Parameter Upgrader QA Device

32.3.2.1 Setting up the keys

The Parameter Upgrader QA Device could be transferring upgrades between peers or transferring upgrades down the heirachy, accordingly the peer to peer Parameter Upgrader QA Device has *three keys (write-parameter* key, *fill/refill* key and *sequence* key) as described in Section 28.6 and

10   Section 26, and a Parameter Upgrader QA Device transferring down the heirachy has *four* keys (*write-parameter* key, *fill/refill* key, *transfer* key and *sequence* Key). These keys must be programmed into the Parameter Upgrader QA Device using the sequence described in Section 32.2.1.

The Key Programming QA Device must be programmed with the appropriate production batch keys

15   , and write-parameter key, fill/refill key, transfer key and sequence key

The GetProgramKey function is called on the Key Programming QA Device with OldKeyRef (OldKeyRef - refer to Section 32.2.1) pointing to a production batch key, and the NewKeyRef (NewKeyRef - refer to Section 32.2.1) pointing to either a write-parameter key, or a fill/refill key, or a transfer key, or a sequence key. The outputs from the GetProgramKey (signature and encrypted

20   New Key) is passed in to ReplaceKey function of the Parameter Upgrader QA Device.

32.3.2.2 Setting up the M0 field in $_{M1}$

The *upgrade value* field and the *count-remaining* field must be defined and set in the upgrade QA Device using the sequence described in Section 32.2.3.

32.3.2.3 Writing upgrade value to the upgrade field

25   The upgrade value is written to upgrade field using the *write-parameter* key. The upgrade QA Device and the trusted QA Device writing to it, share the *write-parameter* key or a variant *write-parameter* key between them i.e $B.K_{n1} = A.K_{n2}$ or $B.K_{n1} = FormKeyVariant(A.K_{n2}, B.ChipId)$, where B is the upgrade QA Device and A is the trusted QA Device. The command sequence used is described in Table 331.

30   *32.3.2.4 Transferring count-remaining amounts*

Finally, the logical count-remaining amounts are transferred to the count-remaining field using the sequence described in Section 31.7.

The QACo will also transfer to the ComCo's upgrade QA Device using the command sequence in Table 331.

35   For a successful transfer from QACo to ComCo, ComCo and QACo must share a *common* key or a *variant* key be i.e $ComCo.K_{n1} = QACo.K_{n2}$ or $ComCo.K_{n1} = FormKeyVariant(QACo.K_{n2}$,ComCo.ChipId). K_{n1} is the *fill/refill* key for the ComCo upgrade QA Device.

32.3.2.5 Setting up sequence data fields

The Parameter Upgrader QA Device has sequence data fields SEQ_1 and SEQ_2 (as described in Section 27) because its count-remaining fields can be refilled as well. These two fields must be initialised to 0xFFFFFFFF, refer to Section 27 for details.

5   The Parameter Upgrader QA Device and the trusted QA Device writing to it, share the *sequence* key or a variant *sequence* key between them i.e $B.K_{n1} = A.K_{n2}$ or $B.K_{n1} = FormKeyVariant(A.K_{n2},$ $B.ChipId)$, where B is the Parameter Upgrader QA Device and A is the trusted QA Device. The command sequence used is described in Table 331.

32.4   SETTING UP THE KEY PROGRAMMER

10   The *key programming QA Device* is set up to replace keys in other QA Devices.

Each *key programming QA Device* must go through the following set up:

- The *key programming QA Device* must be instantiated to factory defaults. Refer to Section 32.1. At the end of instantiation the *key programming QA Device* has production batch keys and no key replacement data.

15   - The *key programming QA Device* must be programmed with the appropriate keys and key replacement map before it can start to replace keys in other QA Devices.

32.4.1   Setting up the keys

The *key programming QA Device* must be programmed with the *key replacement map key*. The *key replacement map key* is described in details in Section 24.

20   The key programming QA Device must programmed with the old and new keys for the QA Devices it is going to perform key replacement on.

Each of the keys is set in the key programming QA Device using the sequence described in Section 32.2.1.

32.4.2   Setting up *key replacement map* field information

25   First the *key replacement map* field information is worked out as per Section 24.1. This field information is set in M1 as per the sequence described Section 32.2.3.

32.4.3   Setting up *key replacement map*

Finally, the *key replacement map* field must be written with the valid mapping using the *key replacement map key*. The key programming QA Device and the trusted QA Device writing to it

30   must share the *key replacement map* key or a variant of *the key replacement map* key between them.

For a successful write of the key replacement map $B.K_{n1} = A.K_{n2}$ or $B.K_{n1} = FormKeyVariant(A.K_{n2},$ $B.ChipId)$, where B is the *key replacement QA Device* and A is the trusted QA Device. The command sequence used is described in Table 331.

35   Appendix A: Field Types

Table 332 lists the field types that are specifically required by the QA Chip Logical Interface and therefore apply across all applications. Additional field types are application specific, and are defined in the relevant application documentation.

Table 332. Predefined Field Types

5

Value	Type	Description
0x0000	0	Non-initialised (default value after final program load)
0x0001	TYPE_PREAUTH	Defines a preauth field in an Ink QA Device
0x0002	TYPE_COUNT_REMAINING	Defines a countRemaining field in an Parameter Upgrader QA Device
0x0003	TYPE_SEQ_1	Defines a sequence data field SEQ_1 in an Ink QA Device or in a Printer QA Device or in an upgrader QA Device
0x0004	TYPE_SEQ_2	Defines a sequence data fields SEQ_2 in an Ink QA Device or in a Printer QA Device or in an upgrader QA Device
0x0005	TYPE_KEY_MAP	Defines a key replacement map in a Key Programmer QA Device
0x0006 and above	reserved	reserved for future use

Appendix B: Key and field definition for different QA Devices

B.1   PARAMETER UPGRADER QA DEVICE

B.1.1   Peer to peer QA Device

10   Table 333. Key definitions for a peer to peer Parameter Upgrader QA Device

Key Name	Purpose
Fill/refill Key	This key has is used for upgrading count-remaining values when the upgrade QA Device is upgraded by another upgrade QA Device and is also used to decrement the count-remaining when upgrading other QA Devices.
Sequence Key	This key is used to initialise sequence data fields SEQ_1 and SEQ_2 to 0xFFFFFFFF.
Write Parameter Key	This key is used to write the upgrade value to the Parameter Upgrader QA Device.

Table 334. Field definitions for a peer to peer Parameter Upgrader QA Device

Field Name	Purpose	Field Attrinutes						
		Type	KeyNum	A [a] RW	NA[b] RW	KPerms[c]	EndPos (Size)	
Count Remaining	The field stores the number of times the Parameter Upgrader QA Device is permitted to upgrade a printer QA Device.	TYPE_COUNT_REMAINING	SN[f] fill/refill key	1	0	KPerms[KN[e]] = 1 Rest are 0	Depends on the maximum number of upgrades that can be stored.	
Upgrade Value	This stores the value that is copied from the Parameter Upgrader QA Device to the field being upgraded on the printer QA Device during the upgrade	Must define the type of the upgrade value i.e TYPE_PRINT_SPEED[d]	SN[f] write-parameter key	1	0	KPerms[KN[e]] = 0 Rest are 0 as well	Set as per upgrade value.	
SEQ_1	This field holds the data for sequence data field SEQ_1 when the Parameter Upgrader QA Device is being upgraded by	TYPE_SEQ_1	SN[f] sequence key	1	0	KPerms[KN[e]] = 0 KPerms[fill /refill[g]] = 1 Rest are 0 as well.	Typically 32 bit.	

	another Parameter Upgrader Refill QA Device.						
SEQ_2	This field holds the data for sequence data fieldsSEQ_2 when the Parameter Upgrader QA Device is being upgraded by another Parameter Upgrader Refill QA Device.	TYPE_SEQ_2	SN[f] sequence key	1	0	KPerms[KN[e]] = 0 KPerms[fill /refill[g]] = 1 Rest are 0 as well.	Typically 32 bit.

a. Authenticated ReadWrite permission

b. Non-authenticated ReadWrite permission

c. KeyPerms

d. This is a sample type only

e. KeyNum

f. Key Slot Number

g. Fill/Refill key has authenticated decrement-only permission to the sequence data fields

B.1.2   Heirarchical Transfer QA Device

Key definitions

Table 335. Key definitions for a Parameter Upgrader QA Device (transferring down the heirachy)

Key Name	Purpose
Transfer Key	This key is used to decrement the count-remaining when upgrading other QA Devices.
Fill/refill Key	This key has is used for upgrading count-remaining values when the Parameter Upgrader QA Device is upgraded by another Parameter

919

	Upgrader QA Device Refill QA Device.
Sequence Key	This key is used to initialise sequence data fields SEQ_1 and SEQ_2 to 0xFFFFFFF.
Write Parameter Key	This key is used to write the upgrade value to the Parameter Upgrader QA Device.

Field definitions

Table 336. Field definitions for Parameter Upgrader QA Device transferring down the hierachy

5

Field Name	Purpose	Field Attrinutes						
		Type	KeyNum	A [a] RW	NA[b] RW	KPerms[c]	EndPo s(Size)
Count Remaining	The field stores the number of times the Parameter Upgrader QA Device is permitted to upgrade a printer QA Device.	TYPE_COUNT_REMAINI NG	SN[f] fill/refill key	1	0	KPerms[KN[e]] = 0 KPerms[Trans fer Key] = 1 Rest are 0	Depen ds on the maxi mum numbe r of upgra des that can be stored.
Upgrade Value	This stores the value that is copied from the Parameter Upgrader QA Device to the field being upgraded on the printer QA	Must define the type of the upgrade value i.e TYPE_PRINT_SPEED[d]	SN[f] write-parameter key	1	0	KeyPerms[K N[e]] = 0 Rest are 0	Set as per upgra de value.

	Device during the upgrade						
SEQ_1	This field holds the data for sequence data fields SEQ_1 when the Parameter Upgrader QA Device is being upgraded by another Parameter Upgrader Refill QA Device.	TYPE_SEQ_1	SN[f] sequence key	1	0	KPerms[KN[e]] = 0 KPerms[fill/refill[g]] = 1 Rest are 0 as well.	Typically 32 bit.
SEQ_2	This field holds the data for sequence data fields SEQ_2 when the Parameter Upgrader QA Device is being upgraded by another Parameter Upgrader Refill QA Device.	TYPE_SEQ_2	SN[f] sequence key	1	0	KPerms[KN[e]] = 0 KPerms[fill/refill[g]] = 1 Rest are 0 as well.	Typically 32 bit.

a. Authenticated ReadWrite permission

b. Non-authenticated ReadWrite permission

c. KeyPerms

d. This is a sample type only

e. KeyNum

f. Key Slot Number

g. Fill/Refill key has authenticated decrement-only permission to the sequence data fields

5

10

B.2 INK REFILL QA DEVICE

B.2.1 Peer to peer QA Device

Key definitions

Table 337. Key definitions for a peer to peer Ink Refill QA Device

5

Key Name	Purpose
Fill/refill Key	This key has is used for filling/refilling ink-remaining values when the Ink Refill QA Device is upgraded by another Ink Refill QA Device and is also used to decrement from the ink-remaining when transferring ink to other QA Devices (typically Ink QA Device).
Sequence Key	This key is used to initialise sequence data fields SEQ_1 and SEQ_2 to 0xFFFFFFF.

Field definitions

Table 338. Field definitions for a peer to peer Ink Refill QA Device

10

Field Name	Purpose	Field Attrinutes					
		Type	Key Num	A ^{a} RW	NA^b RW	KeyPerms^c	EndPos(Size)
Ink Remaining	The field stores the amount of logical ink-remaining in the ink refill QA Device.	Must define the type of Ink e.g TYPE_HIGHQUA LITY_BLACK_IN K^d	SN^f fill/refill key	1	1	KeyPerms[K N^e] = 1 Rest are 0	Depends on the maximum amount of ink that can be stored and the storage resolution i.e in pico litres or in micro litres.
SEQ_1	This field holds the data for	TYPE_SEQ_1	SN^f sequence key	1	0	KPerms[KN^e] = 0	Typically 32 bit.

922

	sequence data field SEQ_1 when the Ink Refill QA Device is being filled/refilled by another Ink Refill QA Device.					KPerms[fill/refill[g]] = 1 Rest are 0 as well.	
SEQ_2	This field holds the data for sequence data field SEQ_2 when the Ink Refill QA Device is being filled/refilled by another Ink Refill QA Device.	TYPE_SEQ_2	SN[f] sequence key	1	0	KPerms[KN[e]] = 0 KPerms[fill/refill[g]] = 1 Rest are 0 as well.	Typically 32 bit.

a. Authenticated ReadWrite permission

b. Non-authenticated ReadWrite permission

c. Decrement-Only For Keys

d. This is a sample type only

e. KeyNum

f. Key Slot Number

g. Fill/Refill key has authenticated decrement-only permission to the sequence data fields

B.2.2 Heirarchical Transfer QA Device

Key definitions

Table 339. Key definitions for a ink refill QA Device (transferring down the heirachy)

Key Name	Purpose
Transfer Key	This key is used to decrement from the ink-remaining when transferring ink to other QA Devices .
Fill/refill Key	This key has is used for filling/refilling ink-remaining values when the Ink Refill QA Device is upgraded by another Ink Refill QA Device.
Sequence Key	This key is used to initialise sequence data fields SEQ_1 and SEQ_2 to 0xFFFFFFFF.

Field definitions

Table 340. Field definitions for a Ink Refill QA Device (transferring down the heirachy)

Field Name	Purpose	Field Attrinutes						
		Type	KeyNum	A [a] RW	NA[b] RW	KeyPerms[c]		EndPos(Size)
Ink Remainin g	The field stores the amount of logical ink-remaining in the Ink Refill QA Device.	Must define the type of Ink e.g- TYPE_HIGHQUALI TY_BLACK_INK[d]	SN[f] fill/refill key	1	0	KPerms[KN[e]] = 0 KPerms[Transfer Key] = 1 Rest are 0		Depends on the maximu m amount of ink that can be stored and the storage resolutio n i.e in pico litres or in micro litres.
SEQ_1	This field holds the data for sequence data field SEQ_1 when the Ink Refill QA Device is being filled/refilled by another Ink Refill QA	TYPE_SEQ_1	SN[f] sequence key	1	0	KPerms[KN[e]] = 0 KPerms[fill/refill[g]] = 1 Rest are 0.		Typicall y 32 bit.

924

	Device.						
SEQ_2	This field holds the data for sequence data field SEQ_2 when the Ink Refill QA Device is being filled/refilled by another Ink Refill QA Device.	TYPE_SEQ_2	SN[f] sequence key	1	0	KPerms[KN[e]] = 0 KPerms[fill/refill[g]] = 1 Rest are 0.	Typically 32 bit.

a. Authenticated ReadWrite permission

b. Non-authenticated ReadWrite permission

c. KeyPerms

d. This is a sample type only

e. KeyNum

f. Key Slot Number

g. Fill/Refill key has authenticated decrement-only permission to the sequence data fields

B.3 KEY PROGRAMMING QA DEVICE

B.3.1 Key definitions

Table 341. Key definitions for a Key Programming QA Device

Key Name	Purpose
Key replacement map Key	This key is used to write the key replacement map.
Old Keys	These are the old keys of the QA Device whose keys will be replaced by the Key Programming QA Device.
New Keys	These are the new keys of the QA Device whose old keys will be replaced by the Key Programming QA Device.

B.3.2 Field definitions

Table 342. Field definitions for a key replacement QA Device

Field Name	Purpose	Field Attrinutes						
		Type	KeyNum	A ^{a} RW	NA^b RW	KPerms^c	EndPos (Size)	
Key replacement map	This defines the mapping between the old key and the new key for the QA Device whose old key will be replaced by the new key.	TYPE_KEY_MAP	Key Replacement Map key	1	0	KPerms[KN^d] = 0 Rest are 0	2 words (64 bits)	

a. Authenticated ReadWrite permission

b. Non-authenticated ReadWrite permission

c. KeyPerms

d. KeyNum

B.4 INK QA DEVICE

B.4.1 Key definitions

Table 343. Key definitions for a Ink QA Device

Key Name	Purpose
Fill/refill Key	This key is used for fill/refilling ink-remaining amount in the ink QA Device.
Ink usage Key	This key is verifying the data read from the ink QA Device and for writing preauth data.
Sequence Key	This key is used to initialise sequence data fields SEQ_1 and SEQ_2 to 0xFFFFFFF.

B.4.2 Field definitions

Table 344. Field definitions for a Ink QA Device

Field Name	Purpose	Field Attrinutes					
		Type	Key Num	A [a] RW	NA [b] RW	KPerms [c]	EndPos (Size)
Ink Remaining	The amount of logical ink-remaining in the ink QA Device. More than one ink-remaining field may be present depending on the number of physical inks stored in the ink cartridge.	Must define the type of Ink i.e TYPE_HQ_BLACK _INK [d]	SN [f] fill/refill key	1	1	KPerms[KN [e]] = 1 Rest are 0	Depends on the maximum amount of ink that can be stored and the storage resolution i.e in pico litres or in micro litres.
Preauth	This field defines the preauth value.	TYPE_PREAUTH	SN [f] ink usage key	0	1	KPerms[KN [e]] = 0 Rest are 0	Depends on preauth amount. Typically 32 bits, may be 64 bits to accomodate larger preauth amounts.
SEQ_1	This field holds the data for sequence data field	TYPE_SEQ_1	SN [f] sequence key	1	0	KPerms[KN [e]] = 0 KPerms[fill/refil	Typically 32 bit.

	SEQ_1 when the Ink QA Device is being filled/refilled by a Ink Refill QA Device.					$]^g] = 1$ Rest are 0.	
SEQ_2	This field holds the data for sequence data field SEQ_2 when the Ink QA Device is being filled/refilled by another Ink Refill QA Device.	TYPE_SEQ_2	SN^f sequence key	1	0	KPerms[KN^e] = 0 KPerms[fill/refil l^g] = 1 Rest are 0.	Typically 32 bit.

a. Authenticated ReadWrite permission

b. Non-authenticated ReadWrite permission

c. KeyPerms

d. This is a sample type only

e. KeyNum

f. Key Slot Number

g. Fill/Refill key has authenticated decrement-only permission to the sequence data fields

B.5   PRINTER QA DEVICE

B.5.1   Key definition

Table 345. Key definitions for a Printer QA Device

Key Name	Purpose
Upgrade key (fill/refill key)	This key is used for writing / upgrading the functional parameter.
Ink usage Key	This key is verifying the data read from the Ink QA Device.
Sequence Key	This key is used to initialise sequence data fields SEQ_1 and SEQ_2 to 0xFFFFFFFF.

PECID/SOPECID Key	This key is used to verify the data read from the printer QA Device. This key is unique to each printer. Also used to translate data from the ink QA Device to the trusted printer system QA Device.

B.5.2   Field definition

Table 346. Field definitions for a Printer QA Device

Field Name	Purpose	Field Attrinutes					
		Type	Key Num	A [a] RW	NA [b] RW	KPerms [c]	EndPos (Size)
Functional parameter	The field stores an upgradeable functional parameter. More than one functional parameter can be stored in the printer QA Device.	Must define the type of print speed i.e TYPE_PRINT_SPEED[d]	SN[f] fill/refill key	1	0	KPerms[KN[e]] = 0 Rest are 0	Set as per functional parameter.
SEQ_1	This field holds the data for sequence data field SEQ_1 when the Printer QA Device is being filled/refilled by a Parameter Upgrader QA Device.	TYPE_SEQ_1	SN[f] sequence key	1	0	KPerms[KN[e]] = 0 KPerms[fill/refill[g]] = 1 Rest are 0.	Typically 32 bit.
SEQ_2	This field holds the data for sequence data field SEQ_2 when the Printer QA Device is being filled/refilled by another Parameter Upgrader	TYPE_SEQ_2	SN[f] sequence key	1	0	KPerms[KN[e]] = 0 KPerms[fill/refill[g]] = 1 Rest are 0.	Typically 32 bit.

	QA Device.						

a. Authenticated ReadWrite permission

b. Non-authenticated ReadWrite permission

c. KeyPerms

5   d. This is a sample type only

e. KeyNum

f. Key Slot Number

g.Fill/Refill key has authenticated decrement-only permission to the sequence data fields

10   B.6   TRUSTED PRINTER SYSTEM QA DEVICE

B.6.1   Key definition

Table 347.

Key Name	Purpose
PECID/SOPECID Key	This key is used to verify the data read from the printer QA Device. This key is unique to each printer. This key is also used for verifying translated data from the ink QA Device.

15

INTRODUCTION

1   Background

This document describes a QA Chip that can be used to hold contains authentication keys together
5   with circuitry specially designed to prevent copying. The chip is manufactured using a standard
Flash memory manufacturing process, and is low cost enough to be included in consumables such
as ink and toner cartridges. The implementation is approximately 1mm^2 in a 0.25 micron flash
process, and has an expected die manufacturing cost of approximately 10 cents in 2003.
Once programmed, the QA Chips as described here are compliant with the NSA export guidelines
10   since they do not constitute a strong encryption device. They can therefore be practically
manufactured in the USA (and exported) or anywhere else in the world.
Note that although the QA Chip is designed for use in authentication systems, it is microcoded, and
can therefore be programmed for a variety of applications.

2   Nomenclature
15   The following symbolic nomenclature is used throughout this document:

Table 348. Summary of symbolic nomenclature

Symbol	Description
F[X]	Function F, taking a single parameter X
F[X, Y]	Function F, taking two parameters, X and Y
X | Y	X concatenated with Y
X ∧ Y	Bitwise X AND Y
X ∨ Y	Bitwise X OR Y (inclusive-OR)
X ⊕ Y	Bitwise X XOR Y (exclusive-OR)
¬X	Bitwise NOT X (complement)
X ← Y	X is assigned the value Y
X ← {Y, Z}	The domain of assignment inputs to X is Y and Z
X = Y	X is equal to Y
X ≠ Y	X is not equal to Y
⇓X	Decrement X by 1 (floor 0)
⇑X	Increment X by 1 (modulo register length)
Erase X	Erase Flash memory register X
SetBits[X, Y]	Set the bits of the Flash memory register X based on Y
Z ← ShiftRight[X, Y]	Shift register X right one bit position, taking input bit from Y and placing the output bit in Z

3 PSEUDOCODE

3.1 Asynchronous

    The following pseudocode:

```
var = expression
```

    means the var signal or output is equal to the evaluation of the expression.

3.2 Synchronous

    The following pseudocode:

```
var ← expression
```

    means the var register is assigned the result of evaluating the expression during this cycle.

3.3 Expression

    Expressions are defined using the nomenclature in Table 348 above. Therefore:

```
var = (a = b)
```

    is interpreted as the var signal is 1 if a is equal to b, and 0 otherwise.

4 DIAGRAMS

Black lines are used to denote data, while red lines are used to denote 1-bit control-signal lines.

LOGICAL INTERFACE

5   Introduction

The QA Chip has a physical and a logical external interface. The physical interface defines how the
QA Chip can be connected to a physical System, while the logical interface determines how that
5   System can communicate with the QA Chip. This section deals with the logical interface.

5.1   OPERATING MODES

The QA Chip has four operating modes - *Idle Mode, Program Mode, Trim Mode* and *Active Mode.*

- *Active Mode* is entered on power-on Reset when the fuse has been blown, and whenever a
  specific authentication command arrives from the System. Program code is only executed in
10   *Active Mode.* When the reset program code has finished, or the results of the command have
  been returned to the System, the chip enters *Idle Mode* to wait for the next instruction.

- *Idle Mode* is used to allow the chip to wait for the next instruction from the System.

- *Trim Mode* is used to determine the clock speed of the chip and to trim the frequency during
  the initial programming stage of the chip (when Flash memory is garbage). The clock
15   frequency *must* be trimmed via Trim Mode *before* Program Mode is used to store the
  program code.

- *Program Mode* is used to load up the operating program code, and is required because the
  operating program code is stored in Flash memory instead of ROM (for security reasons).

Apart from while the QA Chip is executing Reset program code, it is always possible to interrupt the
20   QA Chip and change from one mode to another.

5.1.1   Active Mode

Active Mode is entered in any of the following three situations:

- power-on Reset when the fuse has been blown

- receiving a command consisting of a global id write byte (0x00) followed by the ActiveMode
25   command byte (0x06)

- receiving a command consisting of a local id byte write followed by some number of bytes
  representing opcode and data.

In all cases, Active Mode causes execution of program code previously stored in the flash memory
via Program Mode.

30   If Active Mode is entered by power-on Reset or the global id mechanism, the QA Chip executes
specific reset startup code, typically setting up the local id and other IO specific data. The reset
startup code cannot be interrupted except by a power-down condition. The power-on reset startup
mechanism cannot be used before the fuse has been blown since the QA Chip cannot tell whether
the flash memory is valid or not. In this case the globalid mechanism must be used instead.

35   If Active Mode is entered by the local id mechanism, the QA Chip executes specific code depending
on the following bytes, which function as opcode plus data. The interpretation of the following bytes
depends on whatever software happens to be stored in the QA Chip.

5.1.2 Idle Mode

The QA Chip starts up in *Idle Mode* when the fuse has not yet been blown, and returns to *Idle Mode* after the completion of another mode. When the QA Chip is in *Idle Mode*, it waits for a command from the master by watching the low speed serial line for an id that matches either the global id (0x00), or the chip's local id.

- If the primary id matches the global id (0x00, common to all QA Chips), and the following byte from the master is the Trim Mode id byte, and the fuse has not yet been blown, the QA Chip enters *Trim Mode* and starts counting the number of internal clock cycles until the next byte is received. Trim Mode cannot be entered if the fuse has been blown.
- If the primary id matches the global id (0x00, common to all QA Chips), and the following byte from the master is the Program Mode id byte, and the fuse has not yet been blown, the QA Chip enters *Program Mode*. Program Mode cannot be entered if the fuse has been blown.
- If the primary id matches the global id (0x00, common to all QA Chips), and the following byte from the master is the Active Mode id bytes, the QA Chip enters *Active Mode* and executes startup code, allowing the chip to set itself into a state to subsequently receive authentication commands (includes setting a local id and a trim value).
- If the primary id matches the chip's local id, the QA Chip enters *Active Mode*, allowing the subsequent command to be executed.

The valid 8-bit serial mode values sent after a global id are as shown in Table 349:

Table 349. Command byte values to place chip in specific mode

Value	Interpretation
10101011 (0xAB)	Trim Mode (only functions when the fuse has not been blown)
10001101 (0xAD)	Program Mode (only functions when the fuse has not been blown)
00000110 (0x06)	Active Mode (resets the chip & loads the localId)

5.1.3 Trim Mode

Trim Mode is enabled by sending a global id byte (0x00) followed by the Trim Mode command byte (0xAB). Trim Mode can only be entered while the fuse has not yet been blown.

The purpose of Trim Mode is to set the trim value (an internal register setting) of the internal ring oscillator so that Flash erasures and writes are of the correct duration. This is necessary due to the 2:1 variation of the clock speed due to process variations. If writes an erasures are too long, the Flash memory will wear out faster than desired, and in some cases can even be damaged. Note

that the 2:1 variation due to temperature still remains, so the effective operating speed of the chip is 7-14 MHz around a nominal 10MHz.

Trim Mode works by measuring the number of system clock cycles that occur inside the chip from the receipt of the Trim Mode command byte until the receipt of a data byte. When the data byte is

5   received, the data byte is copied to the trim register and the current value of the count is transmitted to the outside world.

Once the count has been transmitted, the QA Chip returns to *Idle Mode*.

At reset, the internal trim register setting is set to a known value r. The external user can now perform the following operations:

10   • send the global id+write followed by the Trim Mode command byte

• send the 8-bit value v over a specified time t

• send a stop bit to signify no more data

• send the global id+read followed by the Trim Mode command byte

• receive the count c

15   • send a stop bit to signify no more data

At the end of this procedure, the trim register will be v, and the external user will know the relationship between external time t and internal time c. Therefore a new value for v can be calculated.

The Trim Mode procedure can be repeated a number of times, varying both t and v in known ways,

20   measuring the resultant c. At the end of the process, the final value for v is established (and stored in the trim register for subsequent use in Program Mode). This value v must also be written to the flash for later use (every time the chip is placed in Active Mode for the first time after power-up).

For more information about the internal workings of Trim Mode and the accuracy of trim in the QA Chip, see Section 11.2 on page 967.

25   5.1.4   Program Mode

Program Mode is enabled by sending a global id byte (0x00) followed by the Program Mode command byte.

If the QA Chip knows already that the fuse has been blown, it simply does not enter Program Mode. If the QA Chip does not know the state of the fuse, it determines whether or not the internal fuse

30   has been blown by reading 32-bit word 0 of the information block of flash memory. If the fuse has been blown the remainder of data from the Program Mode command is ignored, and the QA Chip returns to *Idle Mode*.

If the fuse is still intact, the chip enters Program Mode and erases the entire contents of Flash memory. The QA Chip then validates the erasure. If the erasure was successful, the QA Chip

35   receives up to 4096 bytes of data corresponding to the new program code and variable data. The bytes are transferred in order $byte_0$ to $byte_{4095}$.

Once all bytes of data have been loaded into Flash, the QA Chip returns to *Idle Mode*.

935

Note that Trim Mode functionality *must be performed before* a chip enters Program Mode for the first time. Otherwise the erasure and write durations could be incorrect.

Once the desired number of bytes have been downloaded in Program Mode, the LSS Master must wait for 80µs (the time taken to write two bytes to flash at nybble rates) before sending the new

5   transaction (e.g. Active Mode). Otherwise the last nybbles may not be written to flash.

5.1.5   After Manufacture

Directly after manufacture the flash memory will be invalid and the fuse will not have been blown. Therefore power-on-reset will not cause Active Mode. Trim Mode must therefore be entered first, and only after a suitable trim value is found, should Program Mode be entered to store a program.

10   Active Mode can be entered if the program is known to be valid.

LOGICAL VIEW OF CPU

6   Introduction

The QA Chip is a 32-bit microprocessor with on-board RAM for scratch storage, on-board flash for program storage, a serial interface, and specific security enhancements.

15   The high level commands that a user of an QA Chip sees are all implemented as small programs written in the CPU instruction set.

The following sections describe the memory model, the various registers, and the instruction set of the CPU.

7   Memory Model

20   The QA Chip has its own internal memory, broken into the following conceptual regions:

- *RAM variables* (3Kbits = 96 entries at 32-bits wide), used for scratch storage (e.g. HMAC-SHA1 processing).

- *Flash memory* (8Kbytes main block + 128 bytes info block) used to hold the non-volatile authentication variables (including program keys etc), and program code. Only 4 KBytes + 64

25   bytes is visible to the program addressing space due to shadowing. Shadowing is where half of each byte is used to validate and verify the other half, thus protecting against certain forms of physical and logical attacks. As a result, two bytes are read to obtain a single byte of data (this happens transparently).

7.1   RAM

30   The RAM region consists of 96 × 32-bit words required for the general functioning of the QA Chip, *but only during the operation of the chip*. RAM is volatile memory: once power is removed, the values are lost. Note that in actual fact memory retains its value for some period of time after power-down, but cannot be considered to be available upon power-up. This has issues for security that are addressed in other sections of this document.

35   RAM is typically used for temporary storage of variables during chip operation. Short programs can also be stored and executed from the RAM.

RAM is addressed from 0 to 5F. Since RAM is in an unknown state upon a RESET (RstL), program code should not assume the contents to be 0. Program code can, however, set the RAM to be a particular known state during execution of the reset command (guaranteed to be received before any other commands).

5   7.2   FLASH VARIABLES

The flash memory region contains the non-volatile information in the QA Chip. Flash memory retains its value after a RESET or if power is removed, and can be expected to be unchanged when the power is next turned on.

Byte 0 of main memory is the first byte of the program run for the command dispatcher. Note that
10   the command dispatcher is always run with shadows enabled.

Bytes 0-7 of the information block flash memory is reserved as follows:

- byte 0-3 = fuse. A value of 0x5555AAAA indicates that the fuse has been blown (think of a physical fuse whose wire is no longer intact).

- bytes 4-7 = random number used to XOR all data for RAM and flash memory accesses

15   After power-on reset (when the fuse is blown) or upon receipt of a globalId Active command, the 32-bit data from bytes 4-7 in the information block of Flash memory is loaded into an internal ChipMask register. In Active Mode (the chip is executing program code), all data read from the flash and RAM is XORed with the ChipMask register, and all data written to the flash and RAM is XORed with the ChipMask register before being written out. This XORing happens completely transparently to the
20   program code. Main flash memory byte 0 onward is the start of program code. Note that byte 0 onward needs to be valid after being XORed with the appropriate bytes of ChipMask.

Even though CPU access is in 8-bit and 32-bit quantities, the data is actually stored in flash a nybble-at-a-time. Each nybble write is written as a byte containing 4 sets of b/¬b pairs. Thus every byte write to flash is writing a nybble to real and shadow. A write mask allows the individual
25   targetting of nybble-at-a-time writes.

The checking of flash vs shadow flash is automatically carried out each read (each byte contains both flash and shadow flash). If all 8 bits are 1, the byte is considered to be in its erased form[1], and returns 0 as the nybble. Otherwise, the value returned for the nybble depends on the size of the overall access and the setting of bit 0 of the 8-bit WriteMask.

30   - All 8-bit accesses (i.e. instruction and program code fetches) are checked to ensure that each byte read from flash is 4 sets of b/¬b pairs. If the data is not of this form, the chip hangs until a new command is issued over the serial interface.

- With 32-bit accesses (i.e. data used by program code), each byte read from flash is checked to ensure that it is 4 sets of b/¬b pairs. A setting of $WriteMask_0 = 0$ means that if the data is

---

[1]TSMC's flash memory has an erased state of all 1s

not valid, then the chip will hang until a new command is issued over the serial interface. A setting of $WriteMask_0 = 1$ means that each invalid nybble is replaced by the upper nybble of the WriteMask. This allows recovery after a write or erasure is interrupted by a power-down.

8   Registers

A number of registers are defined for use by the CPU. They are used for control, temporary storage, arithmetic functions, counting and indexing, and for I/O.

These registers do not need to be kept in non-volatile (Flash) memory. They can be read or written without the need for an erase cycle (unlike Flash memory). Temporary storage registers that contain secret information still need to be protected from physical attack by Tamper Prevention and Detection circuitry and parity checks.

All registers are cleared to 0 on a RESET. However, program code should not assume any RAM contents have any particular state, and should set up register values appropriately. In particular, at the startup entry point, the various address registers need to be set up from unknown states.

8.1   GO

A 1-bit GO register is 1 when the program is executing, and 0 when it is not. Programs can clear the GO register to halt execution of program code once the command has finished executing.

8.2   ACCUMULATOR AND Z FLAG

The Accumulator is a 32-bit general-purpose register that can be thought of as the single data register. It is used as one of the inputs to all arithmetic operations, and is the register used for transferring information between memory registers.

The Z register is a 1-bit flag, and is updated each time the Accumulator is written to. The Z register contains the zero-ness of the Accumulator. $Z = 1$ if the last value written to the Accumulator was 0, and 0 if the last value written was non-0.

Both the Accumulator and Z registers are directly accessible from the instruction set.

8.3   ADDRESS REGISTERS

8.3.1   Program Counter Array and Stack Pointer

A 12-level deep 12-bit Program Counter Array (PCA) is defined. It is indexed by a 4-bit Stack Pointer (SP). The current Program Counter (PC), containing the address of the currently executing instruction, is effectively PCA[SP]. A single register bit, PCRamSel determines whether the program is executing from flash or RAM (0 = flash, 1 = RAM).

The PC is affected by calling subroutines or returning from them, and by executing branching instructions. The SP is affected by calling subroutines or returning from them. There is no bounds checking on calling too many subroutines: the oldest entry in the execution stack will be lost.

The entry point for program code is defined to be address 0 in Flash. This entry point is used whenever the master signals a new transaction.

8.3.2   A0-A3

There are 4 8-bit address registers Each register has an associated memory mode bit designating the address as in Flash (0) or RAM (1).

When an An register is pointing to an address in RAM, it holds the word number. When it is pointing to an address in Flash, it points to a set of 32-bit words that start at a 128-bit (16 byte) alignment. The A0 register has a special use of direct offset e.g. access is possible to (A0),0-7 which is the 32-bit word pointed to by A0 offset by the specified number of words.

8.3.3   WriteMask

The WriteMask register is used to determine how many nybbles will be written during a 32-bit write to Flash, and whether or not an invalid nybble will be replaced during a read from Flash.

During writes to flash, bit n (of 8) determines whether nybble n is written. The unit of writing is a nybble since half of each byte is used for shadow data. A setting of 0xFF means that all 32-bits will be written to flash (as 8 sets of nybble writes).

During 32-bit reads from flash (occurs as 8 reads), the value of WriteMask$_0$ is used to determine whether a read of invalid data is replaced by the upper nybble of WriteMask. If 0, a read of invalid data *is not* replaced, and the chip hangs until a new command is issued over the serial interface. If 1, a read of invalid data *is* replaced by the upper nybble of the WriteMask.

Thus a WriteMask setting of 0 (reset setting) means that no writes will occur to flash, and all reads are not replaced (causing the program to hang if an invalid value is encountered).

8.4   COUNTERS

A number of special purpose counters/index registers are defined:

Table 350. Counter/Index registers

Name	Register Size	Bits	Description
C1	1 × 3	3	Counter used to index arrays and general purpose counter
C2	1 × 6	6	General purpose counter and can be used to index arrays

All these counter registers are directly accessible from the instruction set. Special instructions exist to load them with specific values, and other instructions exist to decrement or increment them, or to branch depending on the whether or not the specific counter is zero.

There are also 2 special flags (not registers) associated with C1 and C2, and these flags hold the zero-ness of C1 or C2. The flags are used for loop control, and are listed here, for although they are not registers, they can be tested like registers.

Table 351. Flags for testing C1 and C2

Name	Description
C1Z	1 = C1 is current zero, 0 = C1 is currently non-zero.
C2Z	1 = C2 is current zero, 0 = C2 is currently non-zero.

8.5 RTMP

The single bit register RTMP allows the implementation of LFSRs and multiple precision shift registers.

During a rotate right (ROR) instruction with operand of RB, the bit shifted out (formally bit 0) is written to the RTMP register. The bit currently in the RTMP register becomes the new bit 31 of the Accumulator. Performing multiple ROR RB commands over several 32-bit values implements a multiple precision rotate/shift right.

The XRB operand operates in the same way as RB, in that the current value in the RTMP register becomes the new bit 31 of the Accumulator. However with the XRB instruction, the bit formally known as bit 0 does not simply replace RTMP (as in the RB instruction). Instead, it is XORed with RTMP, and the result stored in RTMP, thereby allowing the implementation of long LFSRs.

8.6 REGISTERS USED FOR I/O

Several registers are defined for communication between the master and the QA Chip. These registers are LocalId, InByte and OutByte.

LocalId (7 bits) defines the chip-specific id that this particular QA Chip will accept commands for.

InByte (8 bits) provides the means for the QA Chip to obtain the next byte from the master. OutByte (8 bits) provides the means for the QA Chip to send a byte of data to the master.

From the QA Chip's point of view:

- Reads from InByte will hang until there is 1 byte of data present from the master.

- Writes to OutByte will hang if the master has not already consumed the last OutByte.

When the master begins a new command transaction, any existing data in InByte and OutByte is lost, and the PC is reset to the entry point in the code, thus ensuring correct framing of data.

8.7 REGISTERS USED FOR TRIMMING CLOCK SPEED

A single 8-bit Trim register is used to trim the ring oscillaor clock speed. The register has a known value of 0x00 during reset to ensure that reads from flash will succeed at the fastest process corners, and can be set in one of two ways:

- via Trim Mode, which is necessary before the QA Chip is programmed for the first time; or

- via the CPU, which is necessary every time the QA Chip is powered up before any flash write or erasure accesses can be carried out.

8.8 REGISTERS USED FOR TESTING FLASH

There are a number of registers specifically for testing the flash implementation. A single 32-bit write to an appropriate RAM address allows the setting of any combination of these flash test registers.

5   RAM consists of 96 × 32-bit words, and can be pointed to by any of the standard An address registers. A write to a RAM address in the range 97-127 does nothing with the RAM (reads return 0), but a write to a RAM address in the range 0x80-0x87 will write to specific groupings of registers according to the low 3 bits of the RAM address. A 1 in the address bit means the appropriate part of the 32-bit Accumulator value will be written to the appropriate flash test registers. A 0 in the address

10   bit means the register bits will be unaffected.

The registers and address bit groupings are listed in Table 352:

Table 352. Flash test registers settable from CPU in RAM address range 0x80-0x87[2]

adr bitSuperscriptparanumonly	data bits	name	description
0	0	shadowsOff	0 = shadowing applies (nybble based flash access) 1 = shadowing disabled, 8-bit direct accesses to flash.
	1	hiFlashAdr	Only valid when shadowsOff = 1 0 = accesses are to lower 4Kbytes of flash 1 = accesses are to upper 4 Kbytes of flash
	2		
1	3	enableFlashTest	0 = keep flash test register within the TSMC flash IP in its reset state 1 = enable flash test register to take on non-reset values.
	8-4	flashTest	Internal 5-bit flash test register within the TSMC flash IP (SFC008_08B9_HE).

---

[2]   This is from the programmer's perspective. Addresses sent from the CPU are byte aligned, so the MRU needs to test bit n+2. Similarly, checking DRAM address > 128 means testing bit 7 of the address in the CPU, and bit 9 in the MRU.

[3]   unshadowed

[4]   shadowed

			If this is written with 0x1E, then subsequent writes will be according to the TSMC write test mode. You must write a non-0x1E value or reset the register to exit this mode.
2	28-9	flashTime	When timerSel is 1, this value is used for the duration of the program cycle within a standard flash write or erasure. 1 unit = 16 clock cycles (16 × 100ns typical). Regardless of timerSel, this value is also used for the timeout following power down detection before the QA Chip resets itself. 1 unit = 1 clock cycle (= 100ns typical). *Note that this means the programmer should set this to an appropriate value (e.g. 5 µs), just as the localId needs to be set.*
	29	timerSel	0 = use internal (default) timings for flash writes & erasures 1 = use flashTime for flash writes and erasures

When none of the address register bits 0-2 are set (e.g. a write to RAM address 0x80), then invalid writes will clear the illChip and retryCount registers.

For example, set the A0 register to be 0x80 in RAM. A write to (A0),0 will write to none of the flash test registers, but will clear the illChip and retryCount registers. A write to (A0),7 will write to all of the flash test registers. A write to (A0),2 will write to the enableFlashTest and flashTest registers only. A write to (A0),4 will write to the flashTime and timerSel registers etc.

Finally, a write to address 0x88 in RAM will cause a device erasure. If infoBlockSel is 0, then the device erasure will only be of main memory. If infoBlockSel is 1, then the device erasure is of both main memory and the information block (which will also clear the ChipMask and the Fuse).

Reads of invalid RAM areas will reveal information as follows:

- all invalid addresses in RAM (e.g. 0x80) will return the illChip flag in the low bit (illChip is set whenever 16 consecutive bad reads occur for a single byte in memory)
- all invalid addresses in RAM with the low address bit set (e.g. 0x81, or (A0),1 when A0 holds 0x80), will additionally return the most recent retryCount setting (only updated by the chip when a bad read occurs). i.e. bit 0 = illChip, bits 4-1 = retryCount.

8.9   REGISTER SUMMARY

Table 353 provides a summary of the registers used in the CPU.

Table 353. Register summary

Register name	Description	#bits
A[0-3]	address registers	49 =36
Acc	Accumulator	32
C1	general purpose counter and index	3
C2	general purpose counter and index	6
IllChip	gets set whenever more than 15 consecutive bad reads from flash occurred (and any program executing has hung)	1
InByte	input byte from outside world	8
Go	determines whether CPU is executing	1
LocalId	determines id for this chip's IO	7
OutByte	output byte to outside world	8
Z	zero flag for last xfer to Acc	1
PCA	program counter array	1212=144
PCRamSel	Program code is executing in flash (0) or ram (1)	1
RetryCount	counts the number of retries for bad reads	4
RTMP	bit used to alow multi-word rotations	1
SP	stack pointer into PCA	4
Trim	trims ring oscillator frequency	8
flash test registers	various registers in the embedded flash and flash access logic specifically for testing the flash memory	30
TOTAL (bits)		295

8.10 STARTUP

Whenever the chip is powered up, or receives a 'write' command over the serial interface, the PC and PCRamSel get set to 0 and execution begins at 0 in Flash memory. The program (starting at 0) needs to determine how the program was started by reading the InByte register.

If the first byte read is 0xFF, the chip is being requested to perform software reset tasks. Execution of software reset can only be interrupted by a power down. The reset tasks include setting up RAM to contain known startup state information, setting up Trim and localID registers etc. The CPU signals that it is now ready to receive commands from an external device by writing to the OutByte register. An external Master is able to read the OutByte (and any further outbytes that the CPU decides to send) if it so wishes by a read using the localId.

Otherwise the first byte read will be of the form where the least significant bit is 0, and bits 7-1 contain the localId of the device as read over the serial interface. This byte is usually discarded since it nominally only has a value of differentiation against a software reset request. The second and subsequent bytes contain the data message of a write using the localId. The CPU can prevent

5 interruption during execution by writing 0 to the localId and then restoring the desired localId at the later stage.

9   Instruction Set

The CPU operates on 8-bit instructions and typically on 32-bit data items. Each instruction typically consists of an opcode and operand, although the number of bits allocated to opcode and operand

10 varies between instructions.

9.1   BASIC OPCODES (SUMMARY)

The opcodes are summarized in Table 354:

Table 354. Opcode bit pattern map

Opcode	Mnemonic	Simple Description
0000xxxx	JMP	Jump
0001xxxx	JSR	Jump subroutine
0010xxxx	TBR	Test and branch
0011xxxx	DBR	Decrement and branch
0100xxxx	SC	Set counter to a value
0101xxxx	ST	Store Accumulator in specified location
0110000x	-	reserved
01100010	JPZ	Jump to 0
01100011	JPI	Jump indirect
011001xx	-	reserved
01101xxx	-	reserved
01110000	-	reserved
01110001	ERA	Erase page of flash memory pointed to by Accumulator
01110010	JSZ	Jump to subroutine at at 0
01110011	JSI	Jump subroutine indirect
01110100	RTS	Return from subroutine
01110101	HALT	Stop the CPU
0111011x	-	reserved
01111xxx	LIA	Load immediate value into address register
10000xxx	AND	Bitwise AND Accumulator

10001xxx	OR	Bitwise OR Accumulator
1001xxxx	XOR	Exclusive-OR Accumulator
1010xxxx	ADD	Add a 32 bit value to the Accumulator
1011xxxx	LD	Load Accumulator
1100xxxx	ROR	Rotate Accumulator right
11010xxx	AND	Bitwise AND Accumulator[5]
11011xxx	OR	Bitwise OR Accumulator[Superscriptparanumonly]
11100xxx	XOR	Bitwise XOR Accumulator[Superscriptparanumonly]
11101xxx	ADD	Add a 32 bit value to the Accumulator[Superscriptparanumonly]
11110xxx	LD	Load Accumulator[Superscriptparanumonly]
11111xxx	RIA	Rotate Accumulator into address register

Table 355 is a summary of valid operands for each opcode. The table is ordered alphabetically by opcode mnemonic. The binary value for each operand can be found in the subsequent sections.

Table 355. Valid operands for opcodes

5

Opcode	Valid operands
ADD	immediate value (A0), offset (An), {C1,C2} [where n = 0-3]
AND	immediate value (A0), offset
DBR	{C1, C2}, offset
ERA	
HALT	
JMP	address
JPI	
JPZ	
JSI	
JSR	address
JSZ	
LIA	{Flash,Ram}, An [where n = 0-3], {immediate value}
LD	immediate value

---

[5]   immediate form of instruction

	(A0), offset
	(An), {C1,C2} [where n = 0-3]
OR	immediate value
	(A0), offset
RIA	{Flash, Ram}, An [where n = 0-3]
ROR	{InByte, OutByte, WriteMask, ID, C1, C2, RB, XRB, 1,3,8,24,31}
RTS	
SC	{C1, C2}, {immediate value}
ST	(A0), offset
	(An), {C1,C2} [where n = 0-3]
TBR	{0, 1}, offset
XOR	immediate value
	(A0), offset
	(An), {C1,C2} [where n = 0-3]

Additional pseduo-opcodes (for programming convenience) are as follows:

- DEC=ADD 0xFF..
- INC= ADD 0x01
- NOT=XOR 0xFF..
- LDZ = LD 0
- SC {C1, C2}, Acc = ROR {C1, C2}
- RD = ROR Inbyte
- WR = ROR OutByte
- LDMASK = ROR WriteMask
- LDID = ROR Id
- NOP = XOR 0

9.2 ADDRESSING MODES

The CPU supports a set of addressing modes as follows:

- immediate
- accumulator indirect
- indirect fixed
- indirect indexed

9.2.1 Immediate

In this form of addressing, the operand itself supplies the 32-bit data.

Immediate addressing relies on 3 bits of operand, plus an optional 8 bits at PC+1 to determine an 8-bit base value. Bits 0 to 1 of the opcode byte determine whether the base value comes from the opcode byte itself, or from PC+1, as shown in Table 356.

Table 356. Selection for base value in immediate mode

Opcode$_{1-0}$	Base value
00	00000000
01	00000001
10	From PC+1 (i.e. MIUData$_{7-0}$)
11	11111111

The base value is computed by using CMD$_0$ as bit 0, and copying CMD$_1$ into the upper 7 bits.

The resultant 8 bit base value is then used as a 32-bit value, with 0s in the upper 24 bits, or the 8-bit value is replicated into the upper 32 bits. The selection is determined by bit 2 of the opcode byte, as follows:

Table 357. Replicate bits selection

Opcode$_2$	Data
0	No replication. Data has 0 in upper 24 bits and baseVal in lower 8 bits
1	Replicated. Data is 32-bit value formed by replicating baseVal.

Opcodes that support immediate addressing are LD, ADD, XOR, AND, OR. The SC and LIA instructions are also immediate in that they store the data with the opcode, but they are not in the same form as that described here. See the detail on the individual instructions for more information. Single byte examples include:

- LD 0
- ADD 1
- ADD 0xFF... # this subtracts 1 from the acc
- XOR 0xFF... # this performs an effective logical NOT operation

Double byte examples include:

- LD 0x05 # a constant
- AND 0x0F # isolates the lower nybble
- LD 0x36... # useful for HMAC processing

9.2.2   Accumulator indirect

In this form of addressing, the Accumulator holds the effective address.

Opcodes that support Accumulator indirect addressing are JPI, JSI and ERA. In the case of JPI and JSI, the Accumulator holds the address to jump to. In the case of ERA, the Accumulator holds the address of the page in flash memory to be erased.

Examples include:

5
- JPI
- JSI
- ERA

9.2.3 Indirect fixed

In this form of addressing, address register A0 is used as a base address, and then a specific fixed

10 offset is added to the base address to give the effective address.

Bits 2-0 of the opcode byte specify the fixed offset from A0, which means the fixed offset has a range of 0 to 7.

Opcodes that support indirect indexed addressing are LD, ST, ADD, XOR, AND, OR.

Examples include:

15
- LD (A0), 2
- ADD (A0), 3
- AND (A0), 4
- ST (A0), 7

9.2.4 Indirect indexed

20 In this form of addressing, an address register is used as a base address, and then an index register is used to offset from that base address to give the effective address.

The address register is one of 4, and is selected via bits 2-1 of the opcode byte as follows:

Table 358. Address register selection

Opcode$_{2-1}$	address register selected
0 0	A0
0 1	A1
1 0	A2
1 1	A3

25

Bit 0 of the opcode byte selects whether index register C1 or C2 is used:

The counter is selected as follows:

Table 359. Interpretation of counter for DBR

Opcode$_0$	interpretion
0	C1
1	C2

5   Opcodes that support indirect indexed addressing are LD, ST, ADD, XOR.

Examples include:

- LD (A2), C1

- ADD (A1), C1

- ST (A3), C2

10 Since C1 and C2 can only decement, processing of data structures typically works by loading Cn with some number n and decrementing to 0. Thus (Ax),n is the first word accessed, and (Ax),0 is the last 32-bit word accessed in the loop.

9.3   ADD - ADD TO ACCUMULATOR

Mnemonic: ADD

15   Opcode: `1010xxxx, and 11101xxx`

Usage: ADD effective-address, or ADD immediate-value

The ADD instruction adds the specified 32-bit value to the Accumulator via modulo 2^{32} addition.

The `11101xxx` form of the opcode follows the immediate addressing rules (see Section 9.2.1 on page 946). The `1010xxxx` form of the opcode defines an effective address as follows:

20   Table 360. Interpretation of operand for ADD (1010xxxx)

bit 3	interpretion	comment
0	(A0), offset	indirect fixed addressing (see Section 9.2.3 on page 948)
1	(An), Cn	indirect indexed addressing (see Section 9.2.4 on page 948)

The Z flag is also set during this operation, depending on whether the result (loaded into the Accumulator) is zero or not.

25   ### 9.4   AND - BITWISE AND

Mnemonic: AND

Opcode: `10000xxx, and 11010xxx`

Usage: AND effective-address, or AND immediate-value

The AND instruction performs a 32-bit bitwise AND operation on the Accumulator.

949

The `11010xxx` form of the opcode follows the immediate addressing rules (see Section 9.2.1 on page 946). The `10000xxx` form of the opcode follows the indirect fixed addressing rules (see Section 9.2.3 on page 948).

The Z flag is also set during this operation, depending on whether the resultant 32-bit value (loaded into the Accumulator) is zero or not.

9.5 DBR - Decrement and Branch

Mnemonic: DBR

Opcode: `0011xxxx`

Usage: DBR Counter, Offset

This instruction provides the mechanism for building simple loops.

The counter is selected from bit 0 of the opcode byte as follows:

Table 361. Interpretation of counter for DBR

bit 0	interpretion
0	C1
1	C2

If the specified counter is non-zero, then the counter is decremented and the designated offset is added to the current instruction address (PC for 1-byte instructions, PC+1 for 2-byte instructions). If the specified counter is zero, it is decremented (all bits in the counter become set) and processing continues at the next instruction (PC+1 or PC+2). The designated offset will typically be negative for use in loops.

The instruction is either 1 or two bytes, as determined by bits 3-1 of the opcode byte:

- If bits 3-1 = 000, the instruction consumes 2 bytes. The 8 bits at PC+1 are treated as a signed number and used as the offset amount. Thus 0xFF is treated as -1, and 0x01 is treated as +1.

- If bits 3-1 ≠ 000, the instruction consumes 1 byte. Bits 3-1 are treated as a negative number (the sign bit is implied) and used as the offset amount. Thus 111 is treated as -1, and 001 is treated as -7. This is useful for small loops.

The effect is that if the branch is back 1-7 bytes (1 byte is not particularly useful), then the single byte form of the instruction can be used. If the branch is forward, or backward more than 7 bytes, then the 2-byte instruction is required.

9.6 ERA - Erase

Mnemonic: ERA

Opcode: `01110001`

Usage: ERA

This instruction causes an erasure of the 256-byte page of flash memory pointed to by the Accumulator. The Accumulator is assumed to contain an 8-bit pointer to a 128-bit (16 byte) aligned

structure (same structure as the address registers). The page number to be erased comes from bits 7-4, and the lower 4 bits are ignored.

Note that the size of the flash memory page being erased is actually 512 bytes, but in terms of data storage and addressing from the point of view of the CPU, there is only 256 bytes in the page.

5

9.7   HALT - HALT CPU OPERATION

> Mnemonic: HALT
>
> Opcode: 01110101
>
> Usage: HALT

The HALT instruction writes a 0 to the internal GO register, thereby causing the CPU to terminate the

10   currently executing program. The CPU will only be restarted with a new localId transaction from the Master or by a globalId plus Active Mode byte.

9.8   JMP - JUMP

> Mnemonic: JMP
>
> Opcode: 0000xxxx

15   Usage: JMP effective-address

The JMP instruction provides for a method of branching to a specified address. The instruction loads the PC with the effective address.

The new PC is loaded as follows: bits 11-8 are obtained from bits 3-0 of the JMP opcode byte, and bits 7-0 are obtained from PC+1.

20   9.9   JPI - JUMP INDIRECT

> Mnemonic: JPI
>
> Opcode: 01100011
>
> Usage: JPI

The JPI instruction loads the PC with the lower 12 bits of the Accumulator, and sets the PCRamSel

25   register with bit 15 of the Accumulator. Note that the stack is unaffected (unlike JSI).

9.10   JPZ - JUMP TO ZERO

> Mnemonic: JPZ
>
> Opcode: 01100010
>
> Usage: JPZ

30   The JPZ instruction loads the PC and PCRamSel with 0, thereby causing a jump to address 0 in Flash memory.

Programmers will not typically use the JPZ command. However the CPU executes this instruction whenever a new command arrives over the serial interface, so that the code entry point is known i.e. every time the chip receives a new command, execution begins at address 0 in flash. This does

35   not change the status of any other internal register settings (e.g. the flash test registers).

9.11   JSI - JUMP SUBROUTINE INDIRECT

> Mnemonic: JSI

Opcode: 01110011

Usage: JSI

The JSI instruction allows the jumping to a subroutine whose address is obtained from the Accumulator. The instruction pushes the current PC onto the stack, loads the PC with the lower 12 bits of the Accumulator, and sets the PCRamSel register with bit 15 of the Accumulator.

The stack provides for 12 levels of execution (11 subroutines deep). It is the responsibility of the programmer to ensure that this depth is not exceeded or the deepest return value will be overwritten (since the stack wraps). Programs can take advantage of the fact that the stack wraps.

9.12   JSR - JUMP SUBROUTINE

Mnemonic: JSR

Opcode: 0001xxxx

Usage: JSR effective-address

The JSR instruction provides for the most common usage of the subroutine construct. The instruction pushes the current PC onto the stack, and loads the PC with the effective address. The new PC is loaded as follows: bits 11-8 are obtained from bits 3-0 of the JSR opcode byte, and bits 7-0 are obtained from PC+1.

The stack provides for 12 levels of execution (11 subroutines deep). It is the responsibility of the programmer to ensure that this depth is not exceeded or the return value will be overwritten (since the stack wraps). Programs can take advantage of the fact that the stack wraps.

9.13   JSZ - JUMP TO SUBROUTINE AT ZERO

Mnemonic: JSZ

Opcode: 01110010

Usage: JSZ

The JSZ instruction jumps to the subroutine at flash address 0 (i.e. it pushes the current PC onto the stack, and loads the PC and PCRamSel with 0).

Programmers will not typically use the JSZ command. It exists merely as a result of opcode decoding minimization and can be used to assist with the testing of the chip.

9.14   LD - LOAD ACCUMULATOR

Mnemonic: LD

Opcode: 1011xxxx, and 11110xxx

Usage: LD effective-address, or LD immediate-value

The LD instruction loads the Accumulator with the 32-bit value.

The 11110xxx form of the opcode follows the immediate addressing rules (see Section 9.2.1 on page 946). The 1011xxxx form of the opcode defines an effective address as follows:

Table 362. Interpretation of operand for LD (1011xxxx)

bit 3	interpretion	comment
0	(A0), offset	indirect fixed addressing (see Section 9.2.3 on page 948)
1	(An), Cn	indirect indexed addressing (see Section 9.2.4 on page 948)

The Z flag is also set during this operation, depending on whether the value loaded into the Accumulator is zero or not.

9.15   LIA - LOAD IMMEDIATE ADDRESS

Mnemonic: LIA

Opcode: `01111xxx`

Usage: LIAF AddressRegister, Value # for flash addresses

LIAR AddressRegister, Value # for ram addresses

The LIA instruction transfers the data from PC+1 into the designated address register (A0-A3), and sets the memory mode bit for that address register.

Bit 0 specifies whether the address is in flash or ram, as follows:

Table 363. Interpretation of memory mode for LIA

bit 0	interpretion
0	Flash
1	Ram

The address register to be targetted is selected via bits 2-1 of the instruction.

9.16   OR - BITWISE OR

Mnemonic: OR

Opcode: `10001xxx, and 11011xxx`

Usage: OR effective-address, or OR immediate-value

The OR instruction performs a 32-bit bitwise OR operation on the Accumulator.

The `11011xxx` form of the opcode follows the immediate addressing rules (see Section 9.2.1 on page 946). The `10001xxx` form of the opcode follows the indirect fixed addressing rules (see Section 9.2.3 on page 948).

The Z flag is also set during this operation, depending on whether the resultant 32-bit value (loaded into the Accumulator) is zero or not.

9.17   RIA - Rotate In Address

Mnemonic: RIA

Opcode: 11111xxx

Usage: RIAF AddressRegister # for flash addresses

                RIAR AddressRegister # for ram addresses

The RIA instruction transfers the lower 8 bits of the Accumulator into the designated address register (A0-A3), sets the memory mode bit for that address register, and rotates the Accumulator right by 8 bits.

Bit 0 specifies whether the address is in flash or ram, as follows:

Table 364. Interpretation of memory mode for RIA

bit 0	interpretion
0	Flash
1	Ram

The address register to be targetted is selected via bits 2-1 of the instruction.

9.18   ROR - Rotate Right

Mnemonic: ROR

Opcode: 1100xxxx

Usage: ROR Value

The ROR instruction provides a way of rotating the Accumulator right a set number of bits. The bit(s) coming in at the top of the Accumulator (to become bit 31) can either come from the previous lower bits of the Accumulator, from the serial connection, or from external flags. The bit(s) rotated out can also be output from the serial connection, or combined with an external flag.

The allowed operands are as follows:

Table 365. Interpretation of operand for ROR

bits 3-0	interpretion
0000	RB
0001	XRB
0010	WriteMask
0011	1
0100	- (reserved)
0101	3
0110	31
0111	24

954

1000	C1
1001	C2
1010	- (reserved)
1011	- (reserved)
1100	8
1101	ID
1110	InByte
1111	OutByte

The Z flag is also set during this operation, depending on whether resultant 32-bit value (loaded into the Accumulator) is zero or not.

In its simplest form, the operand for the ROR instruction is one of 1, 3, 8, 24, 31, indicating how many bit positions the Accumulator should be rotated. For these operands, there is no external input or output - the bits of the Accumulator are merely rotated right. Note that these values are the equivalent to rotating left 31, 29, 24, 8, 1 bit positions.

With operand WriteMask, the lower 8 bits of the Accumulator are transferred to the WriteMask register, and the Accumulator is rotated right by 1 bit. This conveniently allows successive nybbles to be masked during Flash writes if the Accumulator has been preloaded with an appropriate value (eg 0x01).

With operands C1 and C2, the lower appropriate number of bits of the Accumulator (3 for C1, 6 for C2) are transferred to the C1 or C2 register and the lower 6 bits of the Accumulator are loaded with the previous value of the Cn register. The remaining upper bits of the Accumulator are set as follows: bit 31-24 are copied from previous bits 7-0, and bits 23-6 are copied from previous bits 31-14 (effectively junk). As a result, the Accumulator should be subsequently masked if the programmer wants to compare for specific values).

With operand ID, the 7 low-order bits are transferred from the Accumulator to the LocalId register, the low-order 8 bits of the Accumulator are copied to the Trim register if the Trim register has not already been written to after power-on reset, and the Accumulator is rotated right by 8 bits. This means that the ROR ID instruction needs to be performed twice, typically during Global Active Mode - once to set Trim, and once to set LocalId. *Note there is no way to read the contents of the localId or Trim registers directly.* However the LocalId sent to the program for a command is available as bits 7-1 of the first byte obtained from InByte after program startup.

With operand InByte, the next serial input byte is transferred to the highest 8 bits of the Accumulator. The InByteValid bit is also cleared. If there is no input byte available from the client yet, execution is suspended until there is one. The remainder of the Accumulator is shifted right 8 bit positions (bit31 becomes bit 23 etc.), with lowest bits of the Accumulator shifted out.

With operand OutByte, the Accumulator is shifted right 8 bit positions. The byte shifted out from bits 7-0 is stored in the OutByte register and the OutByteValid flag is set. It is therefore ready for a client to read. If the OutByteValid flag is already set, execution of the instruction stalls until the OutByteValid flag cleared (when the OutByte byte has been read by the client). The new data shifted in to the upper 8

5   bits of the Accumulator is what was transferred to the OutByte register (i.e. from the Accumulator). Finally, the RB and XRB operands allow the implementation of LFSRs and multiple precision shift registers. With RB, the bit shifted out (formally bit 0) is written to the RTMP register. The register currently in the RTMP register becomes the new bit 31 of the Accumulator. Performing multiple ROR RB commands over several 32-bit values implements a multiple precision rotate/shift right. The XRB

10   operates in the same way as RB, in that the current value in the RTMP register becomes the new bit 31 of the Accumulator. However with the XRB instruction, the bit formally known as bit 0 does not simply replace RTMP (as in the RB instruction). Instead, it is XORed with RTMP, and the result stored in RTMP. This allows the implementation of long LFSRs, as required by the authentication protocol.

9.19   RTS - RETURN FROM SUBROUTINE

15
     Mnemonic: RTS

     Opcode: 01110100

     Usage: RTS

The RTS instruction pulls the saved PC from the stack, adds 1, and resumes execution at the resultant address. The effect is to cause execution to resume at the instruction after the most

20   recently executed JSR or JSI instruction.

Although 12 levels of execution are provided for (11 subroutines), it is the responsibility of the programmer to balance each JSR and JSI instruction with an RTS. A RTS executed with no previous JSR will cause execution to begin at whatever address happens to be pulled from the stack. Of course this may be desired behaviour in specific circumstances.

25   ### 9.20   SC - SET COUNTER

     Mnemonic: SC

     Opcode: 0100xxxx

     Usage: SC Counter Value

The SC instruction is used to transfer a 3-bit Value into the specified counter. The operand

30   determines which of counters C1 and C2 is to be loaded as well as the value to be loaded. Value is stored in bits 3-1 of the 8-bit opcode, and the counter is specified by bit 0 as follows:

Table 366. Interpretation of counter for SC

bit 0	interpretion
0	C1
1	C2

Since counter C1 is 3 bits, Value is copied directly into C1.

For counter C2, $C2_{2\text{-}0}$ are copied to $C2_{5\text{-}3}$, and Value is copied to $C2_{2\text{-}0}$. Two SC C2 instructions are therefore required to load C2 with a given 6-bit value. For example, to load C2 with 0x0C, we would have SC C2 1 followed by SC C2 4.

5   9.21   ST - STORE ACCUMULATOR

   Mnemonic: ST

   Opcode: 0101xxxx

   Usage: ST effective-address

The ST instruction stores the 32-bit Accumulator at the effective address. The effective address is
10   determined as follows:

Table 367. Interpretation of operand for ST (0101xxxx)

bit 3	interpretion	comment
0	(A0), offset	indirect fixed addressing (see Section 9.2.3 on page 948)
1	(An), Cn	indirect indexed addressing (see Section 9.2.4 on page 948)

If the effective address in Flash memory, only those nybbles whose corresponding WriteMask bit is
15   set will be written to Flash. Programmers should be very aware of flash characteristics (write time, longevity, page size etc. when storing data in flash).

There is always the possibility that power could be removed during a write to Flash. If this occurs, the flash will be in an indeterminate state. If the QA Chip is warned by the external system that power is about to be removed (via the master causing a transition to Idle Mode), the write will be
20   aborted cleanly at the nearest nybble boundary (writes occur in the order of least significant to most significant).

   9.22   TBR - TEST AND BRANCH

   Mnemonic: TBR

   Opcode: 0010xxxx

25   Usage: TBR Value Offset

The Test and Branch instruction tests the status of the Z flag (the zero-ness of the Accumulator), and then branches if a match ocurs.

The zero-ness is selected from bit 0 of the opcode byte as follows:

Table 368. Interpretation of zero-ness for TBR

bit 0	interpretion
0	true if Acc is zero (Z = 1)
1	true if Acc is non-zero (Z=0)

If the specified zero-test matches, then the designated offset is added to the current instruction address (PC for 1-byte instructions, PC+1 for 2-byte instructions). If the zero-test does not match, processing continues at the next instruction (PC+1 or PC+2). The instruction is either 1 or two bytes, as determined by bits 3-1 of the opcode byte:

- If bits 3-1 = 000, the instruction consumes 2 bytes. The 8 bits at PC+1 are treated as a signed number and used as the offset amount to be added to PC+1. Thus 0xFF is treated as -1, and 0x01 is treated as +1.

- If bits 3-1 ≠ 000, the instruction consumes 1 byte. Bits 3-1 are treated as a positive number (the sign bit is implied) and used as the offset amount to be added to PC. Thus 111 is treated as 7, and 001 is treated as 1. This is useful for skipping over a small number of instructions. The effect is that if the branch is forward 1-7 bytes (1 byte is not particularly useful), then the single byte form of the instruction can be used. If the branch is backward, or forward more than 7 bytes, then the 2-byte instruction is required.

9.23 XOR - BITWISE EXCLUSIVE OR

Mnemonic: XOR

Opcode: `1001xxxx, and 11100xxx`

Usage: XOR effective-address, or XOR immediate-value

The XOR instruction performs a 32-bit bitwise XOR operation on the Accumulator.

The `11100xxx` form of the opcode follows the immediate addressing rules (see Section 9.2.1 on page 946). The `1001xxxx` form of the opcode has an effective address as follows:

Table 369. Interpretation of operand for XOR (1001xxxx)

bit 3	interpretion	comment
0	(A0), offset	indirect fixed addressing (see Section 9.2.3 on page 948)
1	(An), Cn	indirect indexed addressing (see Section 9.2.4 on page 948)

The Z flag is also set during this operation, depending on whether the result (loaded into the Accumulator) is zero or not.

IMPLEMENTATION

10 Introduction

This chapter provides the high-level definition of a CPU capable of implementing the functionality required of an QA Chip.

10.1 PHYSICAL INTERFACE

10.1.1 Pin connections

The pin connections are described in Table 370.

Table 370. Pin connections to QA Chip

pin	direction	description
Vdd	In	Nominal voltage. If the voltage deviates from this by more than a fixed amount, the chip will RESET.
GND	In	
SClk	In	Serial clock
SDa	In/Out	Serial data

The system operating clock SysClk is different to SClk. SysClk is derived from an internal ring oscillator based on the process technology. In the FPGA implementation SysClk is obtained via a 5th pin.

10.1.2 Size and cost

The QA Chip uses a 0.25 µm CMOS Flash process for an area of 1mm^2 yielding a 10 cent manufacturing cost in 2002. A breakdown of area is listed in Table 371.

Table 371. Breakdown of Area for QA Chip

approximate area (mm^2)	description
0.49	8KByte flash memory TSMC: SFC0008_08B9_HE (8K x 8-bits, erase page size = 512 bytes) Area = 724.688µm × 682.05 µm.
0.08	3072 bits of static RAM
0.38	General logic
0.05	Analog circuitry
1	TOTAL (approximate)

Note that there is no specific test circuitry (scan chains or BIST) within the QA Chip (see Section 10.3.10 on page 965), so the total transistor count is as shown in Table 371.

10.1.3 Reset

The chip performs a RESET upon power-up. In addition, tamper detection and prevention circuitry in the chip will cause the chip to either RESET or erase Flash memory (depending on the attack detected) if an attack is detected.

5 ## 10.2 OPERATING SPEED

The base operating system clock SysClk is generated internally from a ring oscillator (process dependant). Since the frequency varies with operating temperature and voltage, the clock is passed through a temperature-based clock filter before use (see Section 10.3.3 on page 961). The frequency is built into the chip during manufacture, and cannot be changed. The frequency is in the

10 range 7-14 MHz.

10.3 GENERAL MANUFACTURING COMMENTS

Manufacturing comments are not normally made when normally describing the architecture of a chip. However, in the case of the QA Chip, the physical implementation of the chip is very much tied to the security of the key. Consequently a number of specialized circuits and components are

15 necessary for implementation of the QA Chip. They are listed here.

- Flash process
- Internal randomized clock
- Temperature based clock filter
- Noise generator

20 - Tamper Prevention and Detection circuitry

- Protected memory with tamper detection
- Boot-strap circuitry for loading program code
- Data connections in polysilicon layers where possible
- OverUnderPower Detection Unit

25 - No scan-chains or BIST

10.3.1 Flash process

The QA Chip is implemented with a standard Flash manufacturing process. It is important that a Flash process be used to ensure that good endurance is achieved (parts of the Flash memory can be erased/written many times).

30 ### 10.3.2 Internal randomized clock

To prevent clock glitching and external clock-based attacks, the operating clock of the chip should be generated internally. This can be conveniently accomplished by an internal ring oscillator. The length of the ring depends on the process used for manufacturing the chip.

Due to process and temperature variations, the clock needs to be trimmed to bring it into a range

35 usable for timing of Flash memory writes and erases.

The internal clock should also contain a small amount of randomization to prevent attacks where light emissions from switching events are captured, as described below.

Finally, the generated clock must be passed through a temperature-based clock filter before being used by the rest of the chip (see Section 10.3.3 on page 961).

The normal situation for FET implementation for the case of a CMOS inverter (which involves a pMOS transistor combined with an nMOS transistor) as shown in Figure 353.

5   During the transition, there is a small period of time where both the nMOS transistor and the pMOS transistor have an intermediate resistance. The resultant power-ground short circuit causes a temporary increase in the current, and in fact accounts for around 20% of current consumed by a CMOS device. A small amount of infrared light is emitted during the short circuit, and can be viewed through the silicon substrate (silicon is transparent to infrared light). A small amount of light is also

10   emitted during the charging and discharging of the transistor gate capacitance and transmission line capacitance.

For circuitry that manipulates secret key information, such information must be kept hidden. Fortunately, IBM's PICA system and LVP (laser voltage probe) both have a requirement for repeatability due to the fact that the photo emissions are extremely weak (one photon requires more

15   than 10^5 switching events). PICA requires around 10^9 pases to build a picture of the optical waveform. Similarly the LVP requires multiple passes to ensure an adequate SNR.

Randomizing the clock stops repeatability (from the point of view of collecting information about the same position in time), and therefore reduces the possibility of this attack.

10.3.3   Temperature based clock filter

20   The QA Chip circuitry is designed to operate within a specific clock speed range. Although the clock is generated by an internal ring oscillator, the speed varies with temperature and power. Since the user supplies the temperature and power, it is possible for an attacker to attempt to introduce race-conditions in the circuitry at specific times during processing. An example of this is where a low temperature causes a clock speed higher than the circuitry is designed for, and this may prevent an

25   XOR from working properly, and of the two inputs, the first may always be returned. These styles of transient fault attacks are documented further in [1]. The lesson to be learned from this is that the input power and operating temperature *cannot be trusted*.

Since the chip contains a specific power filter, we must also filter the clock. This can be achieved with a temperature sensor that allows the clock pulses through only when the temperature range is

30   such that the chip can function correctly.

The filtered clock signal would be further divided internally as required.

10.3.4   Noise Generator

Each QA Chip should contain a noise generator that generates continuous circuit noise. The noise will interfere with other electromagnetic emissions from the chip's regular activities and add noise to

35   the I_{dd} signal. Placement of the noise generator is not an issue on an QA Chip due to the length of the emission wavelengths.

The noise generator is used to generate electronic noise, multiple state changes each clock cycle, and as a source of pseudo-random bits for the Tamper Prevention and Detection circuitry (see Section 10.3.5 on page 962).

A simple implementation of a noise generator is a 64-bit maximal period LFSR seeded with a non-zero number.

10.3.5   Tamper Prevention and Detection circuitry

A set of circuits is required to test for and prevent physical attacks on the QA Chip. However what is actually detected as an attack may not be an intentional physical attack. It is therefore important to distinguish between these two types of attacks in an QA Chip:

- where you *can be certain* that a physical attack has occurred.

- where you *cannot* be certain that a physical attack has occurred.

The two types of detection differ in what is performed as a result of the detection. In the first case, where the circuitry can be certain that a true physical attack has occurred, erasure of flash memory key information is a sensible action. In the second case, where the circuitry cannot be sure if an attack has occurred, there is still certainly something wrong. Action must be taken, but the action should not be the erasure of secret key information. A suitable action to take in the second case is a chip RESET. If what was detected was an attack that has permanently damaged the chip, the same conditions will occur next time and the chip will RESET again. If, on the other hand, what was detected was part of the normal operating environment of the chip, a RESET will not harm the key. A good example of an event that circuitry cannot have knowledge about, is a power glitch. The glitch may be an intentional attack, attempting to reveal information about the key. It may, however, be the result of a faulty connection, or simply the start of a power-down sequence. It is therefore best to only RESET the chip, and not erase the key. If the chip was powering down, nothing is lost. If the System is faulty, repeated RESETs will cause the consumer to get the System repaired. In both cases the consumable is still intact.

A good example of an event that circuitry can have knowledge about, is the cutting of a data line within the chip. If this attack is somehow detected, it could only be a result of a faulty chip (manufacturing defect) or an attack. In either case, the erasure of the secret information is a sensible step to take.

Consequently each QA Chip should have 2 Tamper Detection Lines - one for definite attacks, and one for possible attacks. Connected to these Tamper Detection Lines would be a number of Tamper Detection test units, each testing for different forms of tampering. *In addition, we want to ensure that the Tamper Detection Lines and Circuits themselves cannot also be tampered with.*

At one end of the Tamper Detection Line is a source of pseudo-random bits (clocking at high speed compared to the general operating circuitry). The Noise Generator circuit described above is an adequate source. The generated bits pass through two different paths - one carries the original data, and the other carries the inverse of the data. The wires carrying these bits are in the layer

above the general chip circuitry (for example, the memory, the key manipulation circuitry etc.). The wires must also cover the random bit generator. The bits are recombined at a number of places via an XOR gate. If the bits are different (they should be), a 1 is output, and used by the particular unit (for example, each output bit from a memory read should be ANDed with this bit value). The lines

5   finally come together at the Flash memory Erase circuit, where a complete erasure is triggered by a 0 from the XOR. Attached to the line is a number of triggers, each detecting a physical attack on the chip. Each trigger has an oversize nMOS transistor attached to GND. The Tamper Detection Line physically goes through this nMOS transistor. If the test fails, the trigger causes the Tamper Detect Line to become 0. The XOR test will therefore fail on either this clock cycle or the next one (on

10   average), thus RESETing or erasing the chip.

Figure 349 illustrates the basic principle of a Tamper Detection Line in terms of tests and the XOR connected to either the Erase or RESET circuitry.

The Tamper Detection Line must go through the drain of an output transistor for each test, as illustrated by Figure 350.

15   It is not possible to break the Tamper Detect Line since this would stop the flow of 1s and 0s from the random source. The XOR tests would therefore fail. As the Tamper Detect Line physically passes through each test, it is not possible to eliminate any particular test without breaking the Tamper Detect Line.

It is important that the XORs take values from a variety of places along the Tamper Detect Lines in

20   order to reduce the chances of an attack. Figure 351 illustrates the taking of multiple XORs from the Tamper Detect Line to be used in the different parts of the chip. Each of these XORs can be considered to be generating a ChipOK bit that can be used within each unit or sub-unit.

A typical usage would be to have an OK bit in each unit that is ANDed with a given ChipOK bit each cycle. The OK bit is loaded with 1 on a RESET. If OK is 0, that unit will fail until the next RESET. If

25   the Tamper Detect Line is functioning correctly, the chip will either RESET or erase all key information. If the RESET or erase circuitry has been destroyed, then this unit will not function, thus thwarting an attacker.

The destination of the RESET and Erase line and associated circuitry is very context sensitive. It needs to be protected in much the same way as the individual tamper tests. There is no point

30   generating a RESET pulse if the attacker can simply cut the wire leading to the RESET circuitry. The actual implementation will depend very much on what is to be cleared at RESET, and how those items are cleared.

Finally, Figure 352 shows how the Tamper Lines cover the noise generator circuitry of the chip. The generator and NOT gate are on one level, while the Tamper Detect Lines run on a level above the

35   generator.

10.3.6 Protected memory with tamper detection

It is not enough to simply store secret information or program code in flash memory. The Flash memory and RAM must be protected from an attacker who would attempt to modify (or set) a particular bit of program code or key information. The mechanism used must conform to being used in the Tamper Detection Circuitry (described above).

The first part of the solution is to ensure that the Tamper Detection Line passes directly above each flash or RAM bit. This ensures that an attacker cannot probe the contents of flash or RAM. A breach of the covering wire is a break in the Tamper Detection Line. The breach causes the Erase signal to be set, thus deleting any contents of the memory. The high frequency noise on the Tamper Detection Line also obscures passive observation.

The second part of the solution for flash is to always store the data with its inverse. In each byte, 4 bits contains the data, and 4 bits (the shadow) contains the inverse of the data. If both are 0, this is a valid erase state, and the value is 0. Otherwise, the memory is only valid if the 4 bits of shadow are the inverse of the main 4 bits. The reasoning is that it is possible to add electrons to flash via a FIB, but not take electrons away. If it is possible to change a 0 to 1 for example, it is not possible to do the same to its inverse, and therefore regardless of the sense of flash, an attack can be detected.

The second part of the solution for RAM is to use a parity bit. The data part of the register can be checked against the parity bit (which will not match after an attack).

The bits coming from Flash and RAM can therefore be validated by a number of test units (one per bit) connected to the common Tamper Detection Line. The Tamper Detection circuitry would be the first circuitry the data passes through (thus stopping an attacker from cutting the data lines).

In addition, the data and program code should be stored in different locations for each chip, so an attacker does not know where to launch an attack. Finally, XORing the data coming in and going to Flash with a random number that varies for each chip means that the attacker cannot learn anything about the key by setting or clearing an individual bit that has a probability of being the key (the inverse of the key must also be stored somewhere in flash).

Finally, each time the chip is called, every flash location is read before performing any program code. This allows the flash tamper detection to be activated in a common spot instead of when the data is actually used or program code executed. This reduces the ability of an attacker to know exactly what was written to.

10.3.7 Boot-strap circuitry for loading program code

Program code should be kept in protected flash instead of ROM, since ROM is subject to being altered in a non-testable way. A boot-strap mechanism is therefore required to load the program code into flash memory (flash memory is in an indeterminate state after manufacture).

The boot-strap circuitry must not be in a ROM - a small state-machine suffices. Otherwise the boot code could be trivially modified in an undetectable way.

The boot-strap circuitry must erase all flash memory, check to ensure the erasure worked, and then load the program code.

The program code should only be executed once the flash program memory has been validated via Program Mode.

5   Once the final program has been loaded, a fuse can be blown to prevent further programming of the chip.

10.3.8   Connections in polysilicon layers where possible

Wherever possible, the connections along which the key or secret data flows, should be made in the polysilicon layers. Where necessary, they can be in metal 1, but must never be in the top metal

10   layer (containing the Tamper Detection Lines).

10.3.9   OverUnder Power Detection Unit

Each QA Chip requires an OverUnder Power Detection Unit (PDU) to prevent Power Supply Attacks. A PDU detects power glitches and tests the power level against a Voltage Reference to ensure it is within a certain tolerance. The Unit contains a single Voltage Reference and two

15   comparators. The PDU would be connected into the RESET Tamper Detection Line, thus causing a RESET when triggered.

A side effect of the PDU is that as the voltage drops during a power-down, a RESET is triggered, thus erasing any work registers.

10.3.10   No scan chains or BIST

20   Test hardware on an QA Chip could very easily introduce vulnerabilities. In addition, due to the small size of the QA Chip logic, test hardware such as scan paths and BIST units could in fact take a sizeable chunk of the final chip, lowering yield and causing a situation where an error in the test hardware causes the chip to be unusable. As a result, the QA Chip should not contain any BIST or scan paths. Instead, the program memory must first be validated via the Program Mode

25   mechanism, and then a series of program tests run to verify the remaining parts of the chip.

11   Architecture

Figure 389 shows a high level block diagram of the QA Chip. Note that the tamper prevention and detection circuitry is not shown.

11.1   ANALOGUE UNIT

30   Figure 390 shows a block diagram of the Analogue Unit. Blocks shown in yellow provide additional protection against physical and electrical attack and, depending on the level of security required, may optionally be implemented.

11.1.1   Ring oscillator

The operating clock of the chip (SysClk) is generated by an internal ring oscillator whose frequency

35   can be trimmed to reduce the variation from 4:1 (due to process and temperature) down to 2:1 (temperature variations only) in order to satisfy the timing requirements of the Flash memory.

The length of the ring depends on the process used for manufacturing the chip. A nominal operating frequency range of 10 MHz is sufficient. This clock should contain a small amount of randomization to prevent attacks where light emissions from switching events are captured.

Note that this is different to the input SClk which is the serial clock for external communication.

5 The ring oscillator is covered by both Tamper Detection and Prevention lines so that if an attacker attempts to tamper with the unit, the chip will either RESET or erase all secret information.

FPGA Note: the FPGA does not have an internal ring oscillator. An additional pin (SysClk) is used instead. This is replaced by an internal ring oscillator in the final ASIC.

11.1.2   Voltage reference

10 The voltage reference block maintains an output which is substantially independant of process, supply voltage and temperature. It provides a reference voltage which is used by the PDU and a reference current to stabilise the ring oscillator. It may also be used as part of the temperature based clock filter described in Section 10.3.3 on page 961.

11.1.3   OverUnder power detection unit

15 The OverUnder Power Detection Unit (PDU) is the same as that described in Section 10.3.9 on page 965.

The Under Voltage Detection Unit provides the signal PwrFailing which, if asserted, indicates that the power supply may be turning off. This signal is used to rapidly terminate any Flash write that may be in progress to avoid accidentally writing to an indeterminate memory location.

20 Note that the PDU triggers the RESET Tamper Detection Line only. It does not trigger the Erase Tamper Detection Line.

The PDU can be implemented with regular CMOS, since the key does not pass through this unit. It does not have to be implemented with non-flashing CMOS.

The PDU is covered by both Tamper Detection and Prevention lines so that if an attacker attempts

25 to tamper with the unit, the chip will either RESET or erase all secret information.

11.1.4   Power-on Reset and Tamper Detect Unit

The Power-on Reset unit (POR) detects a power-on condition and generates the PORstL signal that is fed to all the validation units, including the two inside the Tamper Detect Unit (TDU).

All other logic is connected to RstL, which is the PORstL gated by the VAL unit attached to the Reset

30 tamper detection lines (see Section 10.3.5 on page 962) within the TDU. Therefore, if the Reset tamper line is asserted, the validation will drive RstL low, *and can only be cleared by a power-down.* If the tamper line is not asserted, then RstL = PORstL.

The TDU contains a second VAL unit attached to the Erase tamper detection lines (see Section 10.3.5 on page 962) within the TDU. It produces a TamperEraseOK signal that is output to the MIU (1

35 = the tamper lines are all OK, 0 = force an erasure of Flash).

11.1.5 Noise generator

The Noise Generator (NG) is the same as that described in Section 10.3.4 on page 961. It is based on a 64-bit maximal period LFSR loaded with a set non-zero bit pattern on RESET.

The NG must be protected by both Tamper Detection and Prevention lines so that if an attacker attempts to tamper with the unit, the chip will either RESET or erase all secret information.

In addition, the bits in the LFSR must be validated to ensure they have not been tampered with (i.e. a parity check). If the parity check fails, the Erase Tamper Detection Line is triggered.

Finally, all 64 bits of the NG are ORed into a single bit. If this bit is 0, the Erase Tamper Detection Line is triggered. This is because 0 is an invalid state for an LFSR.

11.2 TRIM UNIT

The 8-bit Trim register within the Trim Unit has a reset value of 0x00 (to enable the flash reads to succeed even in the fastest process corners), and is written to either by the PMU during Trim Mode or by the CPU in Active Mode. Note that the CPU is only able to write *once* to the Trim register between power-on-reset due to the TrimDone flag which provides overloading of LocalIdWE.

The reset value of Trim (0) means that the chip has a nominal frequency of 2.7MHz - 10MHz. The upper of the range is when we cannot trim it lower than this (or we could allow some spread on the acceptable trimmed frequency but this will reduce our tolerance to ageing, voltage and temperature which is the range 7MHz to 14MHz). The 2.7MHz value is determined by a chip whose oscillator runs at 10MHz when the trim register is set to its maximum value, so then it must run at 2.7MHz when trim = 0. This is based on the non-linear frequency-current characteristic of the oscillator. Chips found outside of these limits will be rejected.

The frequency of the ring oscillator is measured by counting cycles[6], in the PMU, over the byte period of the serial interface. The frequency of the serial clock, SClk, and therefore the byte period will be accurately controlled during the measurement. The cycle count (Fmeas) at the end of the period is read over the serial bus and the Trim register updated (Trimval) from its power on default (POD) value. The steps are shown in Figure 391. Multiple measure - read - trim cycles are possible to improve the accuracy of the trim procedure.

A single byte for both Fmeas and Trimval provide sufficient accuracy for measurement and trimming of the frequency. If the bus operates at 400kHz, a byte (8 bits) can be sent in 20μs. By dividing the maximum oscillator frequency, expected to be 20MHz, by 2 results in a cycle count of 200 and 50 for the minimum frequency of 5MHz resulting in a worst case accuracy of 2%.

Figure 392 shows a block diagram of the Trim Unit:

---

[6]Note that the PMU counts using 12-bits, saturates at 0xFFF, and returns the cycle count divided by 2 as an 8-bit value. This means that multiple measure-read-trim cycles may be necessary to resolve any amibguity. In any case, multiple cycles are necessary to test the correctness of the trim circuitry during manufacture test.

The 8-bit Trim value is used in the analog Trim Block to adjust the frequency of the ring oscillator by controlling its bias current. The two lsbs are used as a voltage trim, and the 6 msbs are used as a frequency trim.

The analog Trim Clock circuit also contains a Temperature filter as described in Section 10.3.3 on page 961.

11.3   IO UNIT

The QA Chip acts as a *slave* device, accepting serial data from an external master via the IO Unit (IOU). Although the IOU actually transmits data over a 1-bit line, the data is always transmitted and received in 1-byte chunks.

The IOU receives commands from the master to place it in a specific operating mode, which is one of:

- Idle Mode: is the startup mode for the IOU if the fuse has not yet been blown. *Idle Mode* is the mode where the QA Chip is waiting for the next command from the master. Input signals from the CPU are ignored.

- Program Mode: is where the QA Chip erases all currently stored data in the Flash memory (program and secret key information) and then allows new data to be written to the Flash. The IOU stays in *Program Mode* until told to enter another mode.

- Active Mode: is the startup mode for the IOU if the fuse has been blown (the program is safe to run). Active Mode is where the QA Chip allows the program code to be executed to process the master's specific command. The IOU returns to *Idle Mode* automatically when the command has been processed, or if the time taken between consuming input bytes (while the master is writing the data) or generating output bytes (while the master is reading the results) is too great.

- Trim Mode: is where the QA Chip allows the generation and setting of a trim value to be used on the internal ring oscillator clock value. This must be done for safety reasons before a program can be stored in the Flash memory.

See Section 12 on page 970 for detailed information about the IOU.

11.4   CENTRAL PROCESSING UNIT

The Central Processing Unit (CPU) block provides the majority of the circuitry of the 4-bit microprocessor. Figure 393 shows a high level view of the block.

11.5   MEMORY INTERFACE UNIT

The Memory Interface Unit (MIU) provides the interface to flash and RAM. The MIU contains a Program Mode Unit that allows flash memory to be loaded via the IOU, a Memory Request Unit that maps 8-bit and 32-bit requests into multiple byte based requests, and a Memory Access Unit that generates read/write strobes for individual accesses to the memory.

Figure 394 shows a high level view of the MIU block.

11.6 MEMORY COMPONENTS

The Memory Components block isolates the memory implementation from the rest of the QA Chip. The entire contents of the Memory Components block must be protected from tampering. Therefore the logic must be covered by both Tamper Detection Lines. This is to ensure that program code, keys, and intermediate data values cannot be changed by an attacker. The 8-bit wide RAM also needs to be parity-checked.

Figure 395 shows a high level view of the Memory Components block. It consists of 8KBytes of flash memory and 3072 bits of parity checked RAM.

11.6.1 RAM

The RAM block is shown here as a simple 96 × 32-bit RAM (plus parity included for verification). The parity bit is generated during the write.

The RAM is in an unknown state after RESET, so program code cannot rely on RAM being 0 at startup.

The initial version of the ASIC has the RAM implemented by Artisan component RA1SH (96 × 32-bit RAM without parity). Note that the RAMOutEn port is active low i.e. when 0, the RAM is enabled, and when 1, the RAM is disabled.

11.6.2 Flash memory

A single Flash memory block is used to hold all non-volatile data. This includes program code and variables. The Flash memory block is implemented by TSMC component SFC0008_08B9_HE [4], which has the following characteristics:

- 8K × 8-bit main memory, plus 128 × 8-bit information memory
- 512 byte page erase
- Endurance of 20,000 cycles (min)
- Greater than 100 years data retention at room temperature
- Access time: 20 ns (max)
- Byte write time: 20μs (min)
- Page erase time: 20ms (min)
- Device erase time: 200 ms (min)
- Area of 0.494mm^2 (724.66μm × 682.05μm)

The FlashCtrl line are the various inputs on the SFC0008_08B9_HE required to read and write bytes, erase pages and erase the device. A total of 9 bits are required (see [4] for more information). Flash values are unchanged by a RESET. After manufacture, the Flash contents must be considered to be garbage. After an erasure, the Flash contents in the SFC0008_08B9_HE is all 1s.

11.6.3 VAL blocks

The two VAL units are validation units connected to the Tamper Prevention and Detection circuitry (described in Section 10.3.5 on page 962), each with an OK bit. The OK bit is set to 1 on PORstL, and

ORed with the ChipOK values from both Tamper Detection Lines each cycle. The OK bit is ANDed with each data bit that passes through the unit.

In the case of VAL$_1$, the effective byte output from the flash will always be 0 if the chip has been tampered with. This will cause shadow tests to fail, program code will not execute, and the chip will hang.

In the case of VAL$_2$, the effective byte from RAM will always be 0 if the chip has been tampered with, thus resulting in no temporary storage for use by an attacker.

12   I/O Unit

The I/O Unit (IOU) is responsible for providing the physical implementation of the logical interface described in Section 5.1 on page 933, moving between the various modes (Idle, Program, Trim and Active) according to commands sent by the master.

The IOU therefore contains the circuitry for communicating externally with the external world via the SClk and SDa pins. The IOU sends and receives data in 8-bit chunks. Data is sent serially, most significant bit (bit 7) first through to least significant bit (bit 0) last. When a master sends a command to an QA Chip, the command commences with a single byte containing an id in bits 7-1, and a read/write sense in bit 0, as shown in Figure 396.

The IOU recognizes a global id of 0x00 and a local id of LocalId (set after the CPU has executed program code at reset or due to a global id / ActiveMode command on the serial bus). Subsequent bytes contain modal information in the case of global id, and command/data bytes in the case of a match with the local id.

If the master sends data too fast, then the IOU will miss data, since the IOU never holds the bus. The meaning of too fast depends on what is running. In Program Mode, the master must send data a little slower than the time it takes to write the byte to flash (actually written as 2 × 8-bit writes, or 40μs). In ActiveMode, the master is permitted to send and request data at rates up to 500 KHz.

None of the latches in the IOU need to be parity checked since there is no advantage for an attacker to destroy or modify them.

The IOU outputs 0s and inputs 0s if either of the Tamper Detection Lines is broken. This will only come into effect if an attacker has disabled the RESET and/or erase circuitry, since breaking either Tamper Detection Lines should result in a RESET or the erasure of all Flash memory.

The IOU's InByte, InByteValid, OutByte, and OutByteValid registers are used for communication between the master and the QA Chip. InByte and InByteValid provide the means for clients to pass commands and data to the QA Chip. OutByte and OutByteValid provide the means for the master to read data from the QA Chip.

•   Reads from InByte should wait until InByteValid is set. InByteValid will remain clear until the master has written the next input byte to the QA Chip. When the IOU is told (by the FEU or MU) that InByte has been read, the IOU clears the InByteValid bit to allow the next byte to be read from the client.

- Writes to OutByte should wait until OutByteValid is clear. Writing OutByte sets the OutByteValid bit to signify that data is available to be transmitted to the master. OutByteValid will then remain set until the master has read the data from OutByte. If the master requests a byte but OutByteValid is clear, the IOU sends a NAck to indicate the data is not yet ready.

5   When the chip is reset via RstL, the IOU enters ActiveMode to allow the PMU to run to load the fuse. Once the fuse has been loaded (when MIUAvail transitions from 0 to 1) the IOU checks to see if the program is known to be safe. If it is not safe, the IOU reverts to IdleMode. If it is safe (FuseBlown = 1), the IOU stays in ActiveMode to allow the program to load up the localId and do any other reset initialization, and will not process any further serial commands until the CPU has written a byte to

10   the OutByte register (which may be read or not at the discretion of the master using a localId read). In both cases the master is then able to send commands to the QA Chip as described in Section 5.1 on page 933.

Figure 397 shows a block diagram of the IOU.

With regards to InByteValid inputs, set has priority over reset, although both set and reset in correct

15   operation should never be asserted at the same time. With regards to IOSetInByte and IOLoadInByte, if IOSetInByte is asserted, it will set InByte to be 0xFF regardless of the setting of IOLoadInByte.

The two VAL units are validation units connected to the Tamper Prevention and Detection circuitry (described in Section 10.3.5 of the Architecture Overview chapter), each with an OK bit. The OK bit is set to 1 on PORstL, and ORed with the ChipOK values from both Tamper Detection Lines each

20   cycle. The OK bit is ANDed with each data bit that passes through the unit.

In the case of VAL_1, the effective byte output from the chip will always be 0 if the chip has been tampered with. Thus no useful output can be generated by an attacker. In the case of VAL_2, the effective byte input to the chip will always be 0 if the chip has been tampered with. Thus no useful input can be chosen by an attacker.

25   There is no need to verify the registers in the IOU since an attacker does not gain anything by destroying or modifying them.

The current mode of the IOU is output as a 2-bit IOMode to allow the other units within the QA Chip to take correct action. IOMode is defined as shown in Table 372:

Table 372. IOMode values

30

Value	Interpretation
00	Idle Mode
01	Program Mode
10	Active Mode
11	Trim Mode

The Logic blocks generate a 1 if the current IOMode is in Program Mode, Active Mode or Trim Mode respectively. The logic blocks are:

Logic$_1$	IOMode = 01 (Program)
Logic$_2$	IOMode = 10 (Active)
Logic$_3$	IOMode = 11 (Trim)

12.1 STATE MACHINE

There are two state machines in the IOU running in parallel. The first is a byte-oriented state machine, the second is a bit-oriented state machine. The byte-oriented state machine keeps track of the operating mode of the QA Chip while the bit-oriented state machine keeps track of the low-level bit Rx/Tx protocol.

The SDa and SClk lines are connected to the respective pads on the QA Chip. The IOU passes each of the signals from the pads through 2 D-types to compensate for metastability on input, and then a further latch and comparitor to ensure that signals are only used if stable for 2 consecutive internal clock cycles. The circuit is shown in Section 12.1.1 below.

12.1.1   Start/Stop control signals

The StartDetected and StopDetected control signals are generated based upon monitoring SDa synchronized to SClk. The StartDetected condition is asserted on the falling edge of SDa synchronized to SClk, and the StopDetected condition is asserted on the rising edge of SDa synchronized to SClk. In addition we generate feSClk which is asserted on the falling edge of SClk, and reSClk which is asserted on the rising edge of SClk. Finally, feSclkPrev is the value of feSClk delayed by a single cycle.

Figure 398 shows the relationship of inputs and the generation of SDaReg, reSClk, feSClk, feSclkPrev, StartDetected and StopDetected.

The SDaRegSelect logic compensates for the 2:1 variation in clock frequency. It uses the length of the high period of the SClk (from the saturating counter) to select between sda5, sda6 and sda7 as the valid data from 300ns before the falling edge of SClk as follows.

The minimum time for the high period of SClk is 600ns. If the counter <= 4 (i.e. 5 or fewer cycles with SClk = 1) then SDaReg output = sda5 (sample point is equidistant from rising and falling edges). If the counter = 5 or 6 (i.e. 6 or 7 samples where SClk = 1) , then SDaReg output = sda6. If the counter = 7 (the counter saturates when there are 8 samples of SClk = 1), then SDaReg output = sda7. This is shown in pseducode below:

```
If ((counter₂ = 0) ∨ (counter = 4))
        SDaReg = sda5
     ElseIf (counter = 7)
        SDaReg = sda7
     Else
```

```
        SDaReg = sda6
    EndIf
```

The counter also provides a means of enabling start and stop detection. There is a minimum of a 600ns setup and 600ns hold time for start and stop conditions. At 14MHz this means samples 4 and 5 after the rising edge (sample 1 is considered to be the first sample where SClk = 1) could potentially include a valid start or stop condition. At 7 MHz samples 4 and 5 represent 284 and 355ns respectively, although this is after the rising edge of SClk, which itself is 100ns after the setup of data (i.e. 384 and 455ns respectively and therefore safe for sampling). Thus the data will be stable (although not a start or stop). Since we detect stops and starts using sda5 and sda6, we can only validly detect starts and stops 6 cycles after a rising edge, and we need to not-detect starts and stops 4 cycles before the falling edge. We therefore only detect starts and stops when the counter is >= 6 (i.e. when sclk3 and sclk2 are 0 and 1 respectively, sda2 holds sample 1 coincident with the rising edge, sda1 holds sample 2, sda0 holds sample 3, we load the counter with 0 and sample SDa to obtain the new sda0 which will hold sample 4 at the end of the cycle. Thus while the counter is incrementing from 0 to 1, sda0 will hold sample 4. Therefore sample 4 will be in sda6 when the counter is 6.

12.1.2   Control of SDa and SClk pins

The SClk line is always driven by the master. The SDa line is driven low whenever we want to transmit an ACK (SDa is active low) or a 0-bit from OutByte. The generation of the SDa pin is shown in the following pseudocode:

```
TxAck = (bitSM_state = ack) ∧ ((byteSM_state = doWrite) ∨
        (((byteSM_state = getGlobalCmd) ∨ (byteSM_state = checkId)) ∧
AckCmd))

TxBit ← (byteSM_state = doRead) ∧ (bitSM_state = xferBit) ∧
¬OutByte¬bitCount

SDa = ¬(TxAck ∨ TxBit)  # only drive the line when we are xmitting
a 0
```

The slew rate of the SDa line should be restricted to minimise ground bounce. The pad must guarantee a fall time > 20ns. The rise time will be controlled by the external pull up resistor and bus capacitance.

12.1.3   Bit-oriented state machine

The bit-oriented state machine keeps track of the general flow of serial transmission including start/data/ack/stop as shown in the following pseudocode:

```
idle
    EndByte = FALSE
    EndAck = FALSE
    If (StartDetected)
```

```
            state ← starting
        Else
            state ← idle
        EndIf

5

    starting
        EndByte = FALSE
        EndAck = FALSE
        NAck ← 0
10      If (StopDetected)
            state ← idle
        ElseIf (feSClkPrev)
            bitCount ← 0
            state ← xferBit
15      Else
            state ← starting# includes StartDetected
        EndIf

    xferBit
20      EndAck = FALSE
        EndByte = (feSclkPrev ∧ (bitCount = 0))  # after feSclk bitCount
    must be 1..8
        If (feSClk)
            shiftLeft[ioByte, SDaReg] # capture the bit in the ioByte
25  shift register
            bitCount ← bitCount + 1# modulo count due to 3 bit bitCount
        EndIf
        If (StopDetected)
            state ← idle
30      ElseIf (StartDetected)
            state ← starting
        ElseIf (EndByte)
            state ← ack
        Else
35          state ← xferBit
        EndIf
```

```
ack

    EndByte = FALSE

    EndAck = feSclkPrev

    If (StopDetected)

       state ← idle

    ElseIf (StartDetected)

       state ← starting

    ElseIf (EndAck)

       state ← xferBit # bitCount is already 0

    Else

       If (feSClk)

          NAck ← SDaReg         # active low, so 0 = ACK, 1 = NACK

       EndIf

       state ← ack

    EndIf
```

12.1.4   Byte-oriented state machine

The following pseudocode illustrates the general startup state of the IOU and the receipt of a transmission from the master.

```
    rstL    # setup state of registers on reset

       IOMode ← ActiveMode   # to force the fuse to be loaded

       OutByteValid ← 0

       OutByte ← 0

       InByteValid ← 1 # required

       InByte ← 0xFF              # byte = FF = the 'reset' command

       localId ← 0   # loads localId with the globalId so no localId
exists

       state ← wait4fuse

    wait4fuse

       If (MIUAvail)

          If (FuseBlown)#  this  must  be  done  same  cycle  as  seeing
MIUAvail go high

             state ← wait4cpu

          Else

             IOMode ← IdleMode  #  CPU  will  now  require  an  external
ActiveMode to start
```

975

```
                    state ← idle
            Else
                state ← wait4fuse
            EndIf
```

5

```
        wait4cpu
            If (CPUOutByteWE)      # wait for CPU reset activities to finish
                state ← idle              # note: we're still in ActiveMode
            Else
```

10

```
                state ← wait4cpu
            EndIf


        idle
            If (StartDetected)
```

15

```
                state ← checkId
            Else
                state ← idle
            EndIf
```

The first byte received must be checked to ensure it is meant for everyone (globalId of 0) or
20 specifically for us (localId matches). We only send an ACK to a read when there is data available to
send. In addition, writes to the general call address (0) are always ACKed, but reads from the
general call address are only ACKed before the fuse has been blown.

```
        checkId
            isWrite = (ioByte_0 = 0)
```

25

```
            isRead = (ioByte_0 = 1)
            isGlobal = (ioByte_{7-1} = 0)
            globalW = isGlobal ∧ isWrite
            localW = (ioByte_{7-1} = localID) ∧ isWrite ∧ ¬isGlobal
            localR = (ioByte_{7-1} = localID) ∧ isRead ∧ (¬GlobalW ∨
```

30

```
        ¬FuseBlown)
            If (StopDetected)
                state ← idle
            ElseIf (EndByte)
                AckCmd_in = (globalW ∨ localW) ∨ (localR ∧ OutByteValid)
```

35

```
                AckCmd ← AckCmd_in
                If (localW)
```

```
                IOMode ← IdleMode  # jic - any output was pending
                IOOutByteUsed = 1
                IOClearInByte = 1   # ensure there is nothing hanging around
        from before
            EndIf
          ElseIf (EndAck)
            If (globalW)   # globalW and localW are mutually exclusive
                state ← getGlobalCmd
            ElseIf (localW)
                IOMode ← ActiveMode
                IOLoadInByte = 1 # will set inByte to localW (lsb will be
        0)
                state ← doWrite
            ElseIf (localR ∧ IOMode₁ ∧ AckCmd)   # Active  mode  (or  Trim
        when fuse intact)
                state ← doRead
            Else
                state ← idle  # ignore reads unless first in active or
        trim mode
            EndIf
          Else
            state ← checkId
          EndIf
```

With a new global command the IOU waits for the mode byte (see Table page6 on page 934) to determine the new operating mode:

```
        getGlobalCmd
            wantProg = ((ioByte = ProgramModeId) ∧ ¬FuseBlown)
            wantTrim = ((ioByte = TrimModeId) ∧ ¬FuseBlown)
            wantActive = (ioByte = ActiveModeId)
            If (StopDetected)
                state ← idle
            ElseIf (StartDetected)
                state ← checkId
            ElseIf (EndByte)
                AckCmd_in = wantActive ∨ wantProg ∨ wantTrim  # only  ACK  cmds
        we can do
```

```
                AckCmd ← AckCmd_in
                If (AckCmd_in)
                    IOMode ← IdleMode  # jic - any output was pending
                    IOOutByteUsed = 1
                    IOClearInByte = 1   # ensure there is nothing hanging around
            from before
                EndIf
              ElseIf (EndAck)
                If (wantProg)
                    IOMode ← ProgramMode # don't load inByte (we only want the
            data)
                    state ← doWrite
                ElseIf (wantTrim)
                    IOMode ← TrimMode # don't load InByte (we only want the
            next byte)
                    state ← doWrite
                ElseIf (wantActive)   # must be Active
                    IOMode ← ActiveMode
                    IOSetInByte = 1  # 0 for all other cases & states. 1 = sets
            inByte to 0xFF
                    IOLoadInByte = 1 # sets InByteValid (InByte is set to 0xFF
            ('reset' cmd))
                    state ← wait4cpu#  don't  do  anything  til  the  cpu  has
            completed this task
                Else
                    state ← idle  # unknown id, so ignore remainder
                EndIf
              Else
                state ← getGlobalCmd
              EndIf
```
When the master writes bytes to the QA Chip (e.g. parameters for a command), the program must
consume the byte fast enough (i.e. during the sending of the ACK) or subsequent bits may be lost.

The process of receiving bytes is shown in the following pseudocode:

```
doWrite
   If (StopDetected)
      state ← idle                      # stay in whatever IOMode we
were in
   ElseIf (StartDetected)
      state ← checkId
   Else
      If (EndByte)
         IOLoadInByte = ¬InByteValid
      EndIf
      If (EndByte ∧ InByteValid) # will only be when master sends
data too quickly
         state ← idle                    # ACK will not
be sent when in idle state
      Else
         state ← doWrite # ACK will be sent automatically after
byte is Rxed
      EndIf
   EndIf
```

When the master wants to read, the IOU sends one byte at a time as requested. The process is shown in the following pseudocode:

```
doRead
   If (StopDetected)
      state ← idle
   ElseIf (StartDetected)
      state ← checkId
   ElseIf (EndAck)
      If (NAck ∨ ¬OutByteValid)
         state ← idle
      Else
         state ← doRead
      EndIf
   Else
      If (EndByte)
         IOOutByteUsed = 1
```

979

```
        EndIf

    state ← doRead

  EndIf
```

13   Fetch and Execute Unit

13.1   INTRODUCTION

The QA Chip does not require the high speeds and throughput of a general purpose CPU. It must operate fast enough to perform the authentication protocols, but not faster. Rather than have specialized circuitry for optimizing branch control or executing opcodes while fetching the next one (and all the complexity associated with that), the state machine adopts a simplistic view of the world. This helps to minimize design time as well as reducing the possibility of error in implementation.

The FEU is responsible for generating the operating cycles of the CPU, stalling appropriately during long command operations due to memory latency.

When a new transaction begins, the FEU will generate a JPZ (jump to zero) instruction.

The general operation of the FEU is to generate sets of cycles:

- Cycle 0: *fetch cycles*. This is where the opcode is fetched from the program memory, and the effective address from the fetched opcode is generated. The Fetch output flag is set during the final cycle 0 (i.e. when the opcode is finally valid).

- Cycle 1: *execute cycle*. This is where the operand is (potentially) looked up via the generated effective address (from Cycle 0) and the operation itself is executed. The Exec output flag is set during the final cycle 1 (i.e. when the operand is finally valid).

Under normal conditions, the state machine generates multiple Cycle=0 followed by multiple Cycle=1. This is because the program is stored in flash memory, and may take multiple cycles to read. In addition, writes to and erasures of flash memory take differing numbers of cycles to perform. The FEU will stall, generating multiple instances of the same Cycle value with Fetch and Exec both 0 until the input MIURdy = 1, whereupon a Fetch or Exec pulse will be generated in that same cycle.

There are also two cases for stalling due to serial I/O operations:

- The opcode is ROR OutByte, and OutByteValid = 1. This means that the current operation requires outputting a byte to the master, but the master hasn't read the last byte yet.

- The operation is ROR InByte, and InByteValid = 0. This means that the current operation requires reading a byte from the master, but the master hasn't supplied the byte yet.

In both these cases, the FEU must stall until the stalling condition has finished.

Finally, the FEU must stop executing code if the IOU exits Active Mode.

The local Cmd opcode/operand latch needs to be parity-checked. The logic and registers contained in the FEU must be covered by both Tamper Detection Lines. This is to ensure that the instructions to be executed are not changed by an attacker.

13.2 STATE MACHINE

The Fetch and Execute Unit (FEU) is combinatorial logic with the following registers:

Table 373. FEU Registers

Name	#bits	Description
Output registers (visible outside the FEU)		
Cycle	1	0 if the FEU is currently fetching an opcode, 1 if the FEU is currently executing the opcode.
NewMemTrans	1	Is asserted during the start of a potential new memory access. 0 = this is not the first cycle of a set of Cycle 0 or Cycle 1 1 = this is the first cycle of a set of Cycle 0 or Cycle 1 (previous cycle must have been a Fetch or an Exec).
Go	1	1 if the FEU is currently fetching and executing program code (i.e. a program is currently running), 0 if it is not.
Local registers (not visible outside the FEU)		
CurrCmd	8+p	Holds the currently executing instruction (parity checked).
PendingKill	1	The currently executing program is waiting to be halted (waiting due to memory access)
PendingStart	1	A new transaction is waiting to be started (waiting due to memory access or an existing transaction not yet stopped)
WasIdle	1	The previous cycle had an IOMode of IdleMode.

5

In addition, the following externally visible outputs are generated asynchronously:

Table 374. Externally visible asynchronous FEU outputs

Name	#bits	Description
Fetch	1	1 if the FEU is performing the final cycle of a fetch (i.e. Cycle will also be 0). It is set when the NextCmd output is valid. The local Cmd register is latched during the Fetch cycle with either the incoming MIU8Data or an FEU-generated command.

Exec	1	1 if the FEU is performing the final cycle of an execute (i.e. Cycle will also be 1). It is set when the data required by the opcode from the MIU is valid. Other units can execute the Cmd and latch data from the MIU (e.g. from MIUData) during the Exec cycle.
Cmd	8	When Cycle = 0, this holds the next instruction to be executed (during the next Cycle = 1). Is generated based on incoming MIU8Data or substituted FEU command (e.g. JSR 0). When Cycle = 1, this holds the current instruction being executed (based on theCmd).

The Cycle and currCmd registers are not used directly. Instead, their outputs are passed through a VAL unit before use. The VAL units are designed to validate the data that passes through them. Each contains an OK bit connected to both Tamper Prevention and Detection Lines. The OK bit is set to 1 on PORstL, and ORed with the ChipOK values from both Tamper Detection Lines each cycle. The OK bit is ANDed with each data bit that passes through the unit.

In the case of VAL$_1$, the effective Cycle will always be 0 if the chip has been tampered with. Thus no program code will execute.

In the case of VAL$_2$, the effective 8-bit currCmd value will always be 0 if the chip has been tampered with. Multiple 0s will be interpreted as the JSR 0 instruction, and this will effectively hang the CPU. VAL$_2$ also performs a parity check on the bits from currCmd to ensure that currCmd has not been tampered with. If the parity check fails, the Erase Tamper Detection Line is triggered. For more information on Tamper Prevention and Detection circuitry, see Section 10.3.5 on page 962.

13.2.1   Pseudocode

```
reset conditions:
    Fetch = 0
    Exec = 0
    Cycle ← 0
    currCmd ← 0
    Go ← 0
    pendingKill ← 0
    pendingStart ← 0
    newMemTrans ← 0
    wasIdle ← 1 # required to detect if IOU starts in a non-idle
state
```

982

•

The cycle by cycle combinatorial logic behaviour is shown in the following pseudocode:

```
isActive = (IOMode = ActiveMode)
wasIdle ← (IOMode = IdleMode)
```
5
```
wantToStart = (pendingStart ∨ wasIdle) ∧ isActive
newTrans = wantToStart ∧ ¬Go ∧ MIUAvail
pendingStart ← wantToStart ∧ ¬newTrans
killTrans = Go ∧ (¬isActive ∨ pendingKill)
```

10
```
Fetch = newTrans ∨ (Go ∧ ¬Cycle ∧ MIURdy ∧ ¬killTrans)
inDelay = (currCmd = ROR InByte) ∧ ¬InByteValid
outDelay = (currCmd = ROR OutByte) ∧ OutByteValid
ioDelay = inDelay ∨ outDelay
Exec = Go ∧ Cycle ∧ MIURdy ∧ ¬ioDelay
```
15
```
If (Cycle)
   Cmd = currCmd
ElseIf (newTrans)
   Cmd = JPZ # jump to 0
```
20
```
Else
   Cmd = MIU8Data
EndIf
```

```
resetGo = (MIURdy ∧ killTrans) ∨ (Fetch ∧ (Cmd = HALT))
```
25
```
pendingKill ← killTrans ∧ ¬resetGo
```

```
changeCycle = Fetch ∨ Exec          # will only be 1 when Go = 1
Cycle ← newTrans ∨ ((Cycle ⊕ changeCycle) ∧ ¬resetGo)
newMemTrans ← newTrans ∨ (changeCycle ∧ ¬resetGo)
```
30.
```
If (Fetch)
   currCmd ← Cmd
EndIf
```

```
If (resetGo)
```
35
```
   Go ← 0
ElseIf (newTrans)
```

```
                    Go ← 1
                EndIf
14          ALU
```

The Arithmetic Logic Unit (ALU) contains a 32-bit Acc (Accumulator) register as well as the circuitry for simple arithmetic and logical operations.

The logic and registers contained in the ALU must be covered by both Tamper Detection Lines. This is to ensure that keys and intermediate calculation values cannot be changed by an attacker. In addition, the Accumulator must be parity-checked.

A 1-bit Z signal represents the state of zero-ness of the Accumulator. The Accumulator is cleared to 0 upon a RstL, and the Z signal is set to 1. The Accumulator is updated for any of the commands: AND, OR, XOR, ADD, ROR, and RIA, and the Z signal is updated whenever the Accumulator is updated. Note that the Z signal is actually implemented as a nonZ register whose output is passed through an inverter and used as Z.

Each arithmetic and logical block operates on two 32-bit inputs: the current value of the Accumulator, and the current 32-bit output of the DataSel block (either the 32 bit value from MIUData or an immediate value). The AND, OR, XOR and ADD blocks perform the standard 32-bit operations. The remaining blocks are outlined below.

Figure 399 shows a block diagram of the ALU:

The Accumulator is updated for all instructions where the high bit of the opcode is set:

$Logic_1$	$Exec \wedge Cmd_7$

Since the WriteEnables of Acc and nonZ takes Cmd_7 and Exec into account (due to $Logic_1$), these two bits are not required by the multiplexor MX_1 in order to select the output. The output selection for MX_1 only requires bits 6-3 of the Cmd and is therefore simpler as a result (as shown in Table 375).

Table 375. Selection for multiplexor MX_1

	Output	Cmd_{6-3}
MX_1	immOut	$011x \vee 1110$ (LD)
	rorOut	$100x \vee 1111$ (RIA, ROR)
	from XOR	$001x \vee 1100$ (XOR)
	from ADD	$010x \vee 1101$ (ADD)
	from AND	$0000 \vee 1010$ (AND)
	from OR	$0001 \vee 1011$ (OR)

The two VAL units are validation units connected to the Tamper Prevention and Detection circuitry (described in Section 10.3.5 on page 962), each with an OK bit. The OK bit is set to 1 on PORstL, and

ORed with the ChipOK values from both Tamper Detection Lines each cycle. The OK bit is ANDed with each data bit that passes through the unit.

In the case of VAL_1, the effective bit output from the Accumulator will always be 0 if the chip has been tampered with. This prevents an attacker from processing anything involving the Accumulator. VAL_1 also performs a parity check on the Accumulator, setting the Erase Tamper Detection Line if the check fails.

In the case of VAL_2, the effective Z status of the Accumulator will always be true if the chip has been tampered with. Thus no looping constructs can be created by an attacker.

14.1   DATASEL BLOCK

The DataSel block is designed to implement the selection between the MIU32Data and the immediate addressing mode for logical commands.

Immediate addressing relies on 3 bits of operand, plus an optional 8 bits at PC+1 to determine an 8-bit base value. Bits 0 to 1 determine whether the base value comes from the opcode byte itself, or from PC+1, as shown in Table 376.

Table 376. Selection for base value in immediate mode

Cmd_{1-0}	Base value
00	00000000
01	00000001
10	From PC+1 (i.e. $MIUData_{31-24}$)
11	11111111

The base value is computed by using CMD_0 as bit 0, and copying CMD_1 into the upper 7 bits.

The 8-bit base value forms the lower 8 bits of output. These 8 bits are also ANDed with the sense of whether the data is replicated in the upper bits or not (i.e. CMD_2). The resultant bits are copied in 3 times to form the upper 24 bits of the output.

Figure 400 shows a block diagram of the ALU's DataSel block:

14.2   ROR BLOCK

The ROR block implements the ROR and RIA functionality of the ALU.

A 1-bit register named RTMP is contained within the ROR unit. RTMP is cleared to 0 on a RstL, and set during the ROR RB and ROR XRB commands. The RTMP register allows implementation of Linear Feedback Shift Registers with any tap configuration.

Figure 401 shows a block diagram of the ALU's ROR block:

The ROR n, blocks are shown for clarity, but in fact would be hardwired into multiplexor MX_3, since each block is simply a rewiring of the 32-bits, rotated right n bits.

$Logic_1$ is used to provide the WriteEnable signal to RTMP. The RTMP register should only be written to during ROR RB and ROR XRB commands. The combinatorial logic block is:

985

Logic$_1$	Exec $\wedge$ (Cmd$_{7-4}$ = ROR) $\wedge$ (Cmd$_{3-1}$ = 000)

Multiplexor MX$_1$ performs the task of selecting the 6-bit value from Cn instead of bits 13-8 (6 bits) from Acc (the selection is based on the value of Logic$_2$). Bit 5 is required to distinguish ROR from RIA.

Logic$_2$	Cmd$_{5-2}$ = 0x10

Table 377. Selection for multiplexor MX$_1$

	Output	Logic$_2$
MX$_1$	Cn	1
	Acc$_{13-8}$	0

Multiplexor MX$_2$ performs the task of selecting the 8-bit value from InByte instead of the lower 8 bits from the ANDed Acc based on the CMD.

Table 378. Selection for multiplexor MX$_2$

	Output	Cmd$_{4-0}$
MX$_2$	InByte	0x110
	Acc$_{7-0}$	$\neg$(0x110)

Multiplexor MX$_3$ does the final rotating of the 32-bit value. The bit patterns of the CMD operand are taken advantage of:

Table 379. Selection for multiplexor MX$_3$

	Output	Cmd$_{3-0}$	Comments
MX$_3$	ROR 1	00xx	RB, XRB, WriteMask, 1
	ROR 3	010x	3
	ROR 31	0110	31
	ROR 24	0111	24
	ROR 8	1xxx	RIA, InByte, 8, OutByte, C1, C2, ID

14.3   IO Block

The IO block within the ALU implements the logic for communicating with the IOU during instructions that involve the Accumulator. This includes generating appropriate control signals and for generating the correct data for sending during writes to the IOU's OutByte and LocalId registers. Figure 402 shows a block diagram of the IO block:

5     $Logic_1$ is used to provide the LocalIdWE signal to the IOU. The localId register should only be written to during the ROR ID command. Only the lower 7 bits of the Accumulator are written to the localId register.

$Logic_2$ is used to provide the ALUOutByteWE signal to the IOU. The OutByte register should only be written to during the ROR OutByte command. Only the lower 8 bits of the Accumulator are written to

10   the OutByte register.

In both cases we output the lower 8 bits of the Accumulator. The ALUIOData value is ANDed with the output of $Logic_2$ to ensure that ALUIOData is only valid when it is safe to do so (thus the IOU logic never sees the key passing by in ALUIOData). The combinatorial logic blocks are:

$Logic_1$	$Exec \wedge (Cmd_{7-0} = ROR\ ID)$
$Logic_2$	$Exec \wedge (Cmd_{7-0} = ROR\ OutByte)$

15     $Logic_3$ is used to provide the ALUInByteUsed signal to the IOU. The InByte is only used during the ROR InByte command. The combinatorial logic is:

$Logic_3$	$Exec \wedge (Cmd_{7-0} = ROR\ InByte)$

15   Program Counter Unit

The Program Counter Unit (PCU) includes the 12 bit PC (Program Counter), as well as logic for

20   branching and subroutine control.

The PCU latches need to be parity-checked. In addition, the logic and registers contained in the PCU must be covered by both Tamper Detection Lines to ensure that the PC cannot be changed by an attacker.

The PC is implemented as a 12 entry by 12-bit PCA (PC Array), indexed by a 4-bit SP (Stack

25   Pointer) register. The PC, PCRamSel and SP registers are all cleared to 0 on a RstL, and updated during the flow of program control according to the opcodes.

The current value for the PC is normally updated during the Execute cycle according to the command being executed. However it is also incremented by 1 during the Fetch cycle for two byte instructions such as JMP, JSR, DBR, TBR, and instructions that require an additional byte for immediate

30   addressing. The mechanism for calculating the new PC value depends upon the opcode being processed.

Figure 403 shows a block diagram of the PCU:

The ADD block is a simple adder modulo 2^{12} with two inputs: an unsigned 12 bit number and an 8-bit signed number (high bit = sign). The signed input is either a constant of 0x01, or an 8-bit offset (the 8 bits from the MIU).

The "+1" block takes a 4-bit input and increments it by 1 (modulo 12). The "-1" block takes a 4-bit
5   input and decrements it by 1 (modulo 12).

Table 380 lists the different forms of PC control:

Table 381. Different forms of PC control during the Exec cycle

Command	Action
JMP	The PC is loaded with the current 12-bit value as passed in from the MIU.
JPI	The PC is loaded with the current 12-bit value as passed in from the Acc. PCRamSel is loaded with the value from bit 15 of the Acc.
JPZ	The PC is loaded with 0. PCRamSel is loaded with 0 (program in flash)
JSZ	Save old value of PC onto stack for later. The PC is loaded with 0. PCRamSel is loaded with 0 (program in flash).
JSR, JSI	Save old value of PC onto stack for later. The PC is loaded with the current 12-bit value as passed in from either the MIU or the Acc. With JSI, PCRamSel is loaded from the value in bit 15 of the Accumulator.
RTS	Pop old value of PC from stack and increment by 1 to get new PC.
TBR	If the Z flag matches the TBR test, add 8-bit signed number (MIU8Data) to current PC. Otherwise increment current PC by 1.
DBR	If the CZ flag is set, add 8-bit signed offset (MIU8Data) to current PC. Otherwise increment current PC by 1.
All others	Increment current PC by 1

10   The updating of PCRamSel only occurs during JPI, JSI, JPZ and JSZ instructions, detected via $Logic_0$. The same action for the Exec takes place for JMP, JSR, JPI, JSI, JPZ and JSZ, so we specifically detect that case in $Logic_1$. In the same way, we test for the RTS case in $Logic_2$.

$Logic_0$	$Cmd_{7-1} = 011x001$
$Logic_1$	$(Cmd_{7-5} = 000) \lor Logic_0$
$Logic_2$	$Cmd_{7-0} = RTS$

When updating the PC, we must decide if the PC is to be replaced by a completely new value (as in the case of the JMP, JSR, JPI, JSI, JPZ and JSZ instructions), or by the result of the adder (all other instructions). The output from $Logic_1$ ANDed with Cycle can therefore be safely used by the

5   multiplexor to obtain the new PC value (we need to always select PC+1 when Cycle is 0, even though we don't always write it to the PCA).

Note that the JPZ and JSZ instructions are implemented as 12 AND gates that cause the Accumulator value to be ignored, and the new PC to be set to 0. Likewise, the PCRamSel bit is cleared via these two instructions using the same AND mechanism.

10   The input to the 12-bit adder depends on whether we are incrementing by 1 (the usual case), or adding the offset as read from the MIU (when a branch is taken by the DBR and TBR instructions). $Logic_3$ generates the test.

$Logic_3$	Cycle $\wedge$ ((($Cmd_{7\text{-}4}$ = DBR) $\wedge \neg$ CZ) $\vee$ (($Cmd_{7\text{-}4}$ = TBR) $\wedge$ ($Cmd_0 \oplus$ Z)))

The actual offset to be added in the case of the DBR and TBR instructions is either the 8-bit value

15   read from the MIU, or an 8-bit value generated by bits 3-1 of the opcode and treating bit 4 of the opcode as the sign (thereby making DBR immediate branching negative, and TBR immediate branching positive). The former is selected when bits 3-1 of the opcode is 0, as shown by $Logic_4$.

$Logic_4$	If ($Cmd_{3\text{-}1}$ = 000) output MIU8Data
	Else output Cmd_4 | Cmd_4 | Cmd_4 | Cmd_4 | Cmd_4 | $Cmd_{3\text{-}1}$

Finally, the selection of which PC entry to use depends on the current value for SP. As we enter a

20   subroutine, the SP index value must increment, and as we return from a subroutine, the SP index value must decrement. $Logic_1$ tells us when a subroutine is being entered, and $Logic_2$ tells us when the subroutine is being returned from. We use $Logic_2$ to select the altered SP value, but only write to the SP register when Exec and Cmd_4 are also set (to prevent JMP and JPZ from adjusting SP).

The two VAL units are validation units connected to the Tamper Prevention and Detection circuitry

25   (described in Section 10.3.5 on page 962), each with an OK bit. The OK bit is set to 1 on PORstL, and ORed with the ChipOK values from both Tamper Detection Lines each cycle. The OK bit is ANDed with each data bit that passes through the unit. Both VAL units also parity-check the data bits to ensure that they are valid. If the parity-check fails, the Erase Tamper Detection Line is triggered.

In the case of VAL_1, the effective output from the SP register will always be 0. If the chip has been

30   tampered with. This prevents an attacker from executing any subroutines.

In the case of VAL_2, the effective PC output will always be 0 if the chip has been tampered with. This prevents an attacker from executing any program code.

16   Address Generator Unit

The Address Generator Unit (AGU) generates effective addresses for accessing the Memory Unit (MU). In Cycle 0, the PC is passed through to the MU in order to fetch the next opcode. The AGU interprets the returned opcode in order to generate the effective address for Cycle 1. In Cycle 1, the generated address is passed to the MU.

The logic and registers contained in the AGU must be covered by both Tamper Detection Lines. This is to ensure that an attacker cannot alter any generated address. The latches for the counters and calculated address should also be parity-checked.

If either of the Tamper Detection Lines is broken, the AGU will generate address 0 each cycle and all counters will be fixed at 0. This will only come into effect if an attacker has disabled the RESET and/or erase circuitry, since under normal circumstances, breaking a Tamper Detection Line will result in a RESET or the erasure of all Flash memory.

16.1   IMPLEMENTATION

The block diagram for the AGU is shown in Figure 404:

The accessMode and WriteMask registers must be cleared to 0 on reset to ensure that no access to memory occurs at startup of the CPU.

The Adr and accessMode registers are written to during the final cycle of cycle 0 (Fetch) and cycle 1 (Exec) with the address to use during the following cycle phase. For example, when cycle = 1, the PC is selected so that it can be written to Adr during Exec. During cycle 0, while the PC is being output from Adr, the address to be used in the following cycle 1 is calculated (based on the fetched opcode seen as Cmd) and finally stored in Adr when Fetch is 1. The accessMode register is also updated in the same way.

It is important to distinguish between the value of Cmd during different values for Cycle:

- During Cycle 0, when Fetch is 1, the 8-bit input Cmd holds the instruction to be executed in the following Cycle 1. This 8-bit value is used to decode the effective address for the operand of the instruction.

- During Cycle 1, when Exec is 1, Cmd holds the currently executing instruction.

The WriteMask register is only ever written to during execution of an appropriate ROR instruction. Logic$_1$ sets the WriteMask and MMR WriteEnables respectively based on this condition:

Logic$_1$	Exec $\wedge$ (Cmd$_{7\text{-}0}$ = ROR WriteMask)

The data written to the WriteMask register is the lower 8 bits of the Accumulator.

The Address Register Unit is only updated by an RIA or LIA instruction, so the writeEnable is generated by Logic$_2$ as follows:

Logic$_2$	Exec $\wedge$ (Cmd$_{6\text{-}3}$ = 1111)

The Counter Unit (CU) generates counters C1, C2 and the selected N index. In addition, the CU outputs a CZ flag for use by the PCU. The CU is described in more detail below.

The VAL1 unit is a validation unit connected to the Tamper Prevention and Detection circuitry (described in Section 10.3.5 on page 962). It contains an OK bit that is set to 1 on PORstL, and ORed with the ChipOK values from both Tamper Detection Lines each cycle. The OK bit is ANDed with the 12 bits of Adr before they can be used. If the chip has been tampered with, the address output will be always 0, thereby preventing an attacker from accessing other parts of memory. The VAL1 unit also performs a parity check on the Adr Address bits to ensure it has not been tampered with. If the parity-check fails, the Erase Tamper Detection Line is triggered.

16.1.1 Counter Unit

The Counter Unit (CU) generates counters C1 and C2 (used internally). In addition, the CU outputs Cn and flag CZ for use externally. The block diagram for the CU is shown in Figure 405:

Registers C1 and C2 are updated when they are the targets of a DBR, SC or ROR instruction. Logic1 generates the control signals for the write enables as shown in the following pseudocode.

```
isDbrSc = (Cmd_7-4 = DBR) ∨ (Cmd_7-4 = SC)
isRorCn = (Cmd_7-4 = ROR) ∧ (Cmd_3-2 = 10)


CnWE = Exec ∧ (isDbrSc ∨ isRorCn)
C1we = CnWE ∧ ¬Cmd_0
C2we = CnWE ∧ Cmd_0
```

The single bit flag CZ is produced by the NOR of the appropriate C1 or C2 register for use during a DBR instruction. Thus CZ is 1 if the appropriate Cn value = 0.

The actual value written to C1 or C2 depends on whether the ROR, DBR or SC instruction is being executed. During a DBR instruction, the value of either C1 or C2 is decremented by 1 (with wrap). One multiplexor selects between the lower 6 bits of the Accumulator (for ROR instructions), and a 6-bit value for an SC instruction where the upper 3 bits = the low 3 bits from C2, and low 3 bits = low 3 bits from Cmd. *Note that only the lowest 3 bits of the operand are written to C1.*

The two VAL units are validation units connected to the Tamper Prevention and Detection circuitry (described in Section 10.3.5 on page 962), each with an OK bit. The OK bit is set to 1 on PORstL, and ORed with the ChipOK values from both Tamper Detection Lines each cycle. The OK bit is ANDed with each data bit that passes through the unit. All VAL units also parity check the data to ensure the counters have not been tampered with. If a parity check fails, the Erase Tamper Detection Line is triggered.

In the case of VAL1, the effective output from the counter C1 will always be 0 if the chip has been tampered with. This prevents an attacker from executing any looping constructs.

In the case of VAL2, the effective output from the counter C2 will always be 0 if the chip has been tampered with. This prevents an attacker from executing any looping constructs.

16.1.2 Calculate Next Address

This unit generates the address of the operand for the next instruction to be executed. It makes use of the Address Register Unit and PC to obtain base addresses, and the counters from the Counter Unit to assist in generating offsets from the base address.

This unit consists of some simple combinatorial logic, including an adder that adds a 6-bit number to a 10-bit number. The logic is shown in the following pseudocode.

```
isErase = (Cmd_7-0 = ERA)
isSt = (Cmd_7-4 = ST)
isAccRead = (Cmd_7-6 = 10)


# First determine whether this is an immediate mode requiring PC+1
isJmpJsrDbrTbrImmed = (Cmd_7-6 = 00) ∧ (¬Cmd_5 ∨ (Cmd_5-1 = ·1x000))
isLia = (Cmd_7-3 = LIA)
isLogImmed = ((Cmd_7-6 = 11) ∧ ((Cmd_5 ∨ Cmd_4) ∧ (Cmd_5-3 ≠ 111))) ∧
(Cmd_1-0 = 10)
pcSel = Cycle ∨ (¬Cycle ∧ (isJmpJsrDbrTbrImmed ∨ isLogImmed ∨
isLia))


# Generate AnSel signal for the Address Register Unit
A0Sel = (isAccRead ∨ isSt) ∧ (¬Cmd_3 ∨ (Cmd_5-3 = 001))
AnSel_1-0 = ¬A0Sel ∧ Cmd_2-1


# The next address is either the new PC or must be generated
# (we require the base address from Address Register Unit)
nextRAMSel = AnDataOut_8 ∧ ¬isErase
If (nextRAMSel)
    baseAdr = 00 | AnDataOut_7-0 #  ram  addresses  are  already  word
aligned
Else
    baseAdr = AnDataOut_7-0 | 00 # flash addresses are 4-byte aligned
EndIf
# Base address is now word (4-byte) aligned
# Now generate the offset amount to be added to the base address
selCn = (isAccRead ∨ isSt) ∧ (Cmd_5 ∨ Cmd_4) ∧ Cmd_3


offset_0 = (A0Sel ∧ Cmd_0) ∨ (selCn ∧ Cn_0)
offset_1 = (A0Sel ∧ Cmd_1) ∨ (selCn ∧ Cn_1)
```

```
offset_2 = (A0Sel ∧ Cmd_2) ∨ (selCn ∧ Cn_2)
offset_{5-3} = selCn ∧ Cn_{5-3}
If (isErase)
    nextEffAdr_{11-4} = Acc_{7-0}
    nextEffAdr_{3-0} = don't care
Else
    # now we can simply add the offset to the base address to get
    the effective adr
    nextEffAdr_{11-2} = baseAdr + offset # 10 bit plus 6 bit, with wrap
    = 10 bits out
    nextEffAdr_{1-0} = 0 # word access, so lower bits of effadr are 0
EndIf
# Now generate the various signals for use during Cycle=1
# Note that these are only valid when pcSel is 0 (otherwise will
read PC)
nextAccessMode_0 = 1 # want 32-bit access
nextAccessMode_1 = nextRAMSel # ram or flash access (only valid if
rd/wr/erase set)
nextAccessMode_2 = isAccRead  # pcSel takes care of LIA instruction
nextAccessMode_3 = isSt   # write access
nextAccessMode_4 = isErase  # erase page access
```

16.1.3   Address Register Unit

This unit contains 4 × 9-bit registers that are optionally cleared to 0 on PORstL. The 2-bit input AnSel selects which of the 4 registers to output on DataOut. When the writeEnable is set, the AnSel selects which of the 4 registers is written to with the 9-bit DataIn.

17   Program Mode Unit

The Program Mode Unit (PMU) is responsible for Program Mode and Trim Mode operations:

•   Program Mode involves erasing the existing flash memory and loading the new program/data into the flash. The program that is loaded can be a bootstrap program if desired, and may contain additional program code to produce a digital signature of the final program to verify that the program was written correctly (e.g. by producing a SHA-1 signature of the entire flash memory).

•   Trim Mode involves counting the number of internal cycles that have elapsed between the entry of Trim Mode (at the falling edge of the ack) and the receipt of the next byte (at the falling edge of the last bit before the ack) from the Master. When the byte is received, the current count value divided by 2 is transmitted to the Master.

The PMU relies on a fuse (implemented as the value of word 0 of the flash information block) to determine whether it is allowed to perform Program Mode operations. The purpose of this fuse is to prevent easy (or accidental) reprogramming of QA Chips once their purpose has been set. For example, an attacker may want to reuse chips from old consumables. If an attacker somehow

5   bypasses the fuse check, the PMU will still erase all of flash before storing the desired program. Even if the attacker somehow disconnects the erasure logic, they will be unable to store a program in the flash due to the shadow nybbles.

The PMU contains an 8-bit buff register that is used to hold the byte being written to flash and a 12-bit adr register that is used to hold the byte address currently being written to.

10   The PMU is also used to load word 1 of the information block into a 32-bit register (combined from 8-bits of buff, 12-bits of adr, and a further 12-bit register) so it can be used to XOR all data to and from memory (both Flash and RAM) for future CPU accesses. This logic is activated only when the chip enters ActiveMode (so as not to access flash and possibly cause an erasure directly after manufacture since shadows will not be correct). The logic and 32-bit mask register is in the PMU to

15   minimize chip area.

The PMU therefore has an asymmetric access to flash memory:

- writes are to main memory
- reads are from information block memory

The reads and writes are automatically directed appropriately in the MRU.

20   A block diagram of the PMU is shown in Figure 406.

17.1   LOCAL STORAGE AND COUNTERS

The PMU keeps a 1-cycle delayed version of MRURdy, called prevMRURdy. It is used to generate PMNewTrans. Therefore each cycle the PMU performs the following task:

25
        prevMRURdy ← MRURdy ∨ (state = loadByte)

The PMU also requires 1-bit maskLoaded, idlePending and idlePending registers, all of which are cleared to 0 on RstL. The 1-bit fuseBlown register is set to 1 on RstL for security.

17.2   STATE MACHINE

30   The state machine for the PMU is shown in Figure 407, with the pseudocode for the
            various states outlined below.
rstl
        prevMRURdy, maskLoaded, idlePending, adr ← 0 #clear most regs
        fuseBlown ← 1 # for security sake assume the worst

35   state ← idle
            The idle state, entered after reset, simply waits for the IOMode to enter
            ProgramMode, ActiveMode, or TrimMode. Note that the reset value for fuseBlown

means that ProgramMode and TrimMode cannot be entered until after a successful entry into ActiveMode that also clears the fuseBlown register. In state idle, PMEn = ¬maskLoaded, and in state wait4Mode PMEn = 0. In all other states, PMEn = 1.

```
idle
    idlePending ← 0
    PMEn = ¬maskLoaded
    PMNewTrans = 0
    If ((IOMode = ActiveMode) ∧ MRURdy)
        If (maskLoaded)
            state ← wait4mode  # no need to reload mask once loaded
        Else
            adr ← 0   # the location of the fuse is within 32-bit word
0
            state ← loadFuse
        EndIf
    ElseIf ((IOMode = ProgramMode) ∧ MRURdy ∧ ¬fuseBlown) # wait 4
access 2 finish
        maskLoaded ← 0   # the mask is now invalid
        adr ← 0    # the location of the fuse is within 32-bit word 0
        state ← loadFuse
    ElseIf ((IOMode = TrimMode) ∧ MRURdy ∧ ¬fuseBlown) #   wait   4
access 2 finish
        maskLoaded ← 0   # the mask is now invalid
        adr ← 0 # start the counter on entering TrimMode
        state ← trim
    Else
        state ← idle
    EndIf
```

The wait4mode state simply waits until for the current mode to finish and returns to idle.

```
wait4mode
    PMEn = 0
    PMNewTrans = 0
    If (IOMode = IdleMode)
        state ← idle
    Else
```

```
            state ← wait4mode
        EndIf
```

The trim state is where we count the number of cycles between the entry of the Trim Mode and the arrival of a byte from the Master. When the byte arrives from the Master, we send the resultant count:

```
    trim
        # We saturate the adder at all 1s to make external trim control
        easier
        lastOne = adr₀ ∧ adr₁ ∧ ... adr₁₁
        If (¬lastOne)
            adr = adr + 1  # 12 bit incrementor
        EndIf
        # This logic simply causes the current adder value to be written
        to the
        # outByte when the inByte is received. The inByte is cleared
        when received
        # although it is not strictly necessary to do so
        PMOutByteWE = InByteValid  # 0 in all other states
        PMInByteUsed = InByteValid  # same as in loadByte state, 0 in all
        other states
        If (IOMode ≠ TrimMode)
            state ← idle
        ElseIf (InByteValid)
            state ← wait4mode
        Else
            state ← trim
        EndIf
```

The loadFuse state is called whenever there is an attempt to program the device or we are entering ActiveMode and the mask is invalid (i.e. after power up or after a ProgramMode or TrimMode command). We load the 32-bit fuse value from word 0 of information memory in flash and compare it against the FuseSig constant (0x5555AAAA) to obtain the fuse value. The next state depends on IOMode and the Fuse.

```
    loadFuse
        PMEn = 1
        PMNewTrans = prevMRURdy
        idlePending_in = idlePending ∨ (IOMode = IdleMode)
```

```
        idlePending ← idlePending_in
    If (MRURdy)
        If (idlePending_in) # . don't change state until the memory
access is complete
            state ← idle
        Else
            fuseBlown_in = (MRUData$_{31-0}$ = FuseSig)
            fuseBlown ← fuseBlown_in
            If (IOMode = ProgramMode)
                If (fuseBlown_in)
                    state ← wait4mode  # not allowed to program anymore
                Else
                    state ← erase
                EndIf
            Elsif (IOMode = ActiveMode)
                adr ← 4   # byte 4 is word 1 (the location of the
XORMask)
                state ← getMask
            Else
                state ← idle
            EndIf
        EndIf
    Else
        state ← loadFuse
    EndIf
```

The erase state erases the flash memory and then leads into the main programming states:

```
    erase
        PMNewTrans = prevMRURdy
        PMEraseDevice = 1 # is 0 in all other states
        adr ← 0
        idlePending_in = idlePending ∨ (IOMode ≠ ProgramMode)
        idlePending ← idlePending_in
        If (MRURdy)
            If (idlePending_in)
                state ← idle
            Else
```

```
                state ← loadByte
            EndIf
        Else
            state ← erase
        EndIf
```
Program Mode involves loading a series of 8-bit data values into the Flash. The PMU reads bytes via the IOU's InByte and InByteValid, setting MUInByteUsed as it loads data. The Master must send data slightly slower than the speed it takes to write to Flash to ensure that data is not lost.

```
        loadByte              # Load in 1 byte (1 word) from IO Unit
            PMNewTrans = 0
            PMInByteUsed = InByteValid # same as in TrimIn state, and 0 in
            all other states
            If (IOMode ≠ ProgramMode)
                state ← idle
            Else
                If (InByteValid)
                    buff ← InByte
                    state ← writeByte
                Else
                    state ← loadByte
                EndIf
            EndIf


        writeByte
            PMNewTrans = prevMRURdy
            PMRW = 0            # write. In all other states, PMRW = 1 (read)
            PM32Out₇₋₀ = buff    # data (can be tied to this)
            PM32Out₁₉₋₈ = adr # can be tied to this
            PM32Out₃₁₋₂₀ = 12bitReg # is always this (is don't care during a
            write)
            idlePending_in = idlePending ∨ (IOMode ≠ ProgramMode)
            idlePending ← idlePending_in
            If (MRURdy)
                lastOne = adr₀ ∧ adr₁ ∧ ... adr₁₁
                adr ← adr + 1 # 12 bit incrementor
                If (idlePending_in)
```

```
                    state ← idle
                ElseIf (lastOne)
                    state ← wait4Mode
                Else
```
5
```
                    state ← loadByte
                EndIf
            Else
                state ← writeByte
            EndIf
```
10   The getMask state loads up word 1 of the flash information block (bytes 4-7) into the 32-bit buffer so
it can be used to XOR all data to and from memory (both Flash and RAM) for future CPU accesses.
```
            getMask
                PMNewTrans = prevMRURdy
                PM32Out19-8 = adr  # adr should = 4, i.e. word 1 which holds the
```
15
```
            CPU's mask
                PMRW = 1           # read (MUST be 1 in this state)
                idlePending_in = idlePending ∨ (IOMode ≠ ActiveMode)
                idlePending ← idlePending_in
                If (MRURdy)
```
20
```
                    buff ← MRUData7-0
                    adr ← MRUData19-8
                    12bitReg ← MRUData31-20
                    maskLoaded ← 1
                    If (idlePending_in)
```
25
```
                        state ← idle
                    Else
                        state ← wait4mode
                    EndIf
                Else
```
30
```
                    state ← getMask
                EndIf
```
18   Memory Request Unit

The Memory Request Unit (MRU) provides arbitration between PMU memory requests and CPU-
based memory requests.

35   The arbitration is straightforward: if the input PMEn is asserted, then PMU inputs are processed and
CPU inputs are ignored. If PMEn is deasserted, the reverse is true.

A block diagram of the MRU is shown in Figure 408.

18.1 ARBITRATION LOGIC

The arbitration logic block provides arbitration between the accesses of the PM and the 8/32-bit accesses of the CPU via a simple multiplexing mechanism based on PMEn:

```
        ReqDataOut31-8 = CPUDataOut31-8
        If (PMEn)
            NewTrans = PMNewTrans
            AccessMode0 = PMRW  # maps to 1 for reads (32 bits), 0 for
writes (8 bits)
            AccessMode1 = 0  # flash accesses only
            AccessMode2 = PMRW ∧ ¬PMEraseDevice # read has lower priority
than erase
            AccessMode3 = ¬PMRW ∧ ¬PMEraseDevice  #  write  has  lower
priority than erase
            AccessMode4 = 0  # pageErase
            AccessMode5 = PMEraseDevice  # erase everything (main & info
block)
            WriteMask = 0xFF
            Adr = PM32Out19-8
            ReqDataOut7-0 = PM32Out7-0
        Else
            NewTrans = CPUNewTrans ∧ (CPUAccessMode4-2 ≠ 000)
            AccessMode4-0 = CPUAccessMode
            AccessMode5 = 0 # cpu cannot ever erase entire chip
            WriteMask = CPUWriteMask
            Adr = CPUAdr
            ReqDataOut7-0 = CPUDataOut7-0
        EndIf
```

18.2 MEMORY REQUEST LOGIC

The Memory Request Logic in the MRU implements the memory requests from the selected input. An individual request may involve outputting multiple sub-requests e.g. an 8-bit read consists of 2 × 4-bit reads (each flash byte contains a nybble plus its inverse).

The input accessMode bits are interpreted as follows:

Table 382. Interpretation of accessMode bits

Bit	Description
0	0 = 8-bit access

1000

	1 = 32-bit access
1	0 = flash access 1 = RAM access this bit is only valid if bit 2, 3 or 4 is set
2	1 = read access
3	1 = write access
4	1 = erase page access
5	1 = erase entire (info and main) flash (only used within the MRU)

The MRU contains the following registers for general purpose flow control:

Table 383. Description of register settings

name	#bits	Description
ActiveTrans	1	Is there a transaction still running? If so, then extraTrans and nextToXfer can be considered valid.
badUntilRestart	1	0 = memory (flash and ram) reads work correctly 1 = memory (flash and ram) reads return 0 Gets set whenever illChip gets set, and remains set until a soft restart occurs i.e. IOMode passes through Idle.
extraTrans	1	Determines whether there is an additional sub-transaction to perform. e.g. a 32 bit read from flash involves 4 sub-transactions in the case of 8-bit accesses, and 8 sub-transactions in the case of 4-bit accesses.
IllChip	1	0 = 15 consecutive bad reads have not occurred 1 = 15 consecutive bad reads have occurred
nextToXfer	3	The next element (byte or nybble) number to transfer to/from memory
restartPending	1	1 = IOMode passed through Idle while a transaction was being processed 0 = The transaction completed without IOMode passing through Idle
retryCount	4	Number of times that a byte has been read badly

		from flash. When a byte has been read badly 15 consecutive times *illChip* will be set.
retryStarted	1	0 = no retries encountered yet for this read 1 = retries have been encountered - retryCount holds the number of retries The retryStarted register is used to stop retryCount being cleared on good reads - thus keeping a record of the last number of retries on a bad read.

Table 383 lists the registers specifically for testing flash. Although the complete set of flash test registers is in both the MRU and MAU (group 0 is in the MRU, groups 1 and 2 are in the MAU), all the decoding takes place from the MRU.

5

10

Table 383. Flash test registers settable from CPU when the RAM address is > 128^7

adr bitSuperscriptparanum only	bits	name	description
0	0	shadowsOff	0 = regular shadowing (nybble based access to flash) 1 = shadowing disabled, 8-bit direct accesses to flash.
	1	hiFlashAdr	Only valid when shadowsOff = 1 0 = accesses are to lower 4Kbytes of flash 1 = accesses are to upper 4 Kbytes of flash
	2		

---

[7]   This is from the programmer's perspective. Addresses sent from the CPU are byte aligned, so the MRU needs to test bit n+2. Similarly, checking DRAM address > 128 means testing bit 7 of the address in the CPU, and bit 9 in the MRU.

1	3	enableFlashTest	0 = keep flash test register within the TSMC flash IP in its reset state 1 = enable flash test register to take on non-reset values.
	8-4	flashTest	Internal 5-bit flash test register within the TSMC flash IP (SFC008_08B9_HE). If this is written with 0x1E, then subsequent writes will be according to the TSMC write test mode. You must write a non-0x1E value or reset the register to exit this mode.
2	28-9	flashTime	When timerSel is 1, this value is used for the duration of the program cycle within a standard flash write or erasure. 1 unit = 16 clock cycles (16 × 100ns typical). Regardless of timerSel, this value is also used for the timeout following power down detection before the QA Chip resets itself. 1 unit = 1 clock cycle (= 100ns typical). *Note that this means the programmer should set this to an appropriate value (e.g. 5 μs), just as the localId needs to be set.*
	29	timerSel	0 = use internal (default) timings for flash writes & erasures 1 = use flashTime for flash writes and erasures

18.2.1 Reset

Initialization on reset involves clearing all the flags:

```
MRURdy = 0   # can't process anything at this point
activeTrans ← 0
extraTrans ← 0
illChip ← 0
badUntilRestart ← 0
restartPending ← 0
```

5

---

[8]   unshadowed

[9]   shadowed

```
          retryCount ← 0

          retryStarted ← 0

          nextToXfer ← 0   # don't care

          shadowsOff ← 0
```
5
```
          hiFlashAdr ← 0

          infoBlockSel ← 0# used to generate MRUMode₂
```

18.2.2 Main logic

The main logic consists of waiting for a new transaction, and starting an appropriate sub-transaction accordingly, as shown in the following pseudocode:

10
```
          #   Generate   some   basic   signals   for   use   in   determining
          accessPatterns

          Is32Bit = AccessMode₀

          Is8Bit = ¬AccessMode₀

          IsFlash = ¬AccessMode₁
```
15
```
          IsRAM = AccessMode₁

          IsRead = AccessMode₂

          IsWrite = AccessMode₃

          noShadows = shadowsOff

          doShadows = IsFlash ∧ ¬noShadows
```
20
```
          continueRequest = (IOMode ≠ IdleMode)

          okForTrans = ¬restartPending ∧ continueRequest

          startOfSubTrans = (NewTrans ∨ extraTrans) ∧ okForTrans

          doingTrans = startOfSubTrans ∨ (activeTrans ∧ ¬extraTrans)

          IsInvalidRAM = doingTrans ∧ IsRAM ∧ (Adr₉ ∨ (Adr₈ ∧ Adr₇))
```
25
```
          IsTestModeWE = doingTrans ∧ IsRAM ∧ IsWrite ∧ Adr₉

          IsTestReg₀ = IsTestModeWE ∧ Adr₃   #write to flash test register -
          bit 1 of word adr

          IsTestReg₁ = IsTestModeWE ∧ Adr₄   #write to flash test register -
          bit 2 of word adr
```
30
```
          MRUTestWE = IsTestReg₀ ∨ IsTestReg₁

          IsPageErase = AccessMode₄

          IsDeviceErase = AccessMode₅ ∨ (IsTestModeWE ∧ (Adr₈₋₂ = 0001000)) #
          bit 9 not req

          IsErase = IsDeviceErase ∨ IsPageErase
```
35
```
          MRURAMSel = IsRAM ∧ ¬MRUTestWE ∧ ¬IsDeviceErase
```

```
IsInfBlock = (PMEn ∧ (IsDeviceErase ∨ IsRead)) ∨
        (¬PMEn ∧ infoBlockSel ∧
            (IsDeviceErase ∨ (IsFlash ∧ (Adr₁₁₋₇ = 0) ∧ ¬(Adr₆ ∧
doShadows))))
```

```
IsInfBlock = (PMEn ∧ (IsDeviceErase ∨ IsRead)) ∨
        (¬PMEn ∧ infoBlockSel ∧
            (IsDeviceErase ∨ (IsFlash ∧ (Adr_{11-7} = 0) ∧ ¬(Adr_6 ∧
doShadows))))
```

# Which element (byte or nybble) are we up to xferring?
If (NewTrans)
   toXfer = 0
Else
   toXfer = nextToXfer
EndIf

# Form the address that goes to the outside world
If (IsFlash ∧ noShadows)
   byteCount = toXfer_{1-0}
   MRUAdr_{12} = hiFlashAdr # upper or lower block of 4Kbytes of flash
   MRUAdr_{11-2} = Adr_{11-2}   # word #
   MRUAdr_{1-0} = (Adr_{1-0} ∧ (¬Is32Bit|¬Is32Bit)) ∨ byteCount # byte
Else
   byteCount = toXfer_{2-1}
   MRUAdr_{12-3} = Adr_{11-2}   # word #
   MRUAdr_{2-1} = (Adr_{1-0} ∧ (¬Is32Bit|¬Is32Bit)) ∨ byteCount # byte
   MRUAdr_0 = toXfer_0 #nybble
EndIf

# Assuming a write, are we allowed to write to this address?
writeEn = SelectBit[WriteMask, ((MRUAdr_2 ∧ doShadows)| MRUAdr_{1-0})] #
mux: 1 from 8

Generate the 4-bit mask to be used for XORing during CPU access
to flash
baseMask = SelectNybble(PM32Out, MRUAdr_{2-0}) # mux selects 4 bits of
32
If (PMEn)
   theMask = 0
Else
   theMask = baseMask # we only use mask for CPU accesses to flash
```

```
        EndIf


        # Select a byte (and nybble) from the data for writes
        baseByte = SelectByte[ReqDataOut, byteCount]  # mux:  8  bits  from
        32
        baseNybble = SelectNybble[baseByte, toXfer$_0$]  # mux: 4 bits from 8
        outNybble = baseNybble $\oplus$ theMask  # only used when nybble writing


        # Generate the data on the output lines (doesn't matter for reads
        or erasures)
        MRUDataOut$_{31-8}$ = ReqDataOut$_{31-8}$ #  effectively  don't  care  for  flash
        writes
        If (doShadows)
            MRUDataOut$_7$ = $\neg$outNybble$_3$
            MRUDataOut$_6$ = outNybble$_3$
            MRUDataOut$_5$ = $\neg$outNybble$_2$
            MRUDataOut$_4$ = outNybble$_2$
            MRUDataOut$_3$ = $\neg$outNybble$_1$
            MRUDataOut$_2$ = outNybble$_1$
            MRUDataOut$_1$ = $\neg$outNybble$_0$
            MRUDataOut$_0$ = outNybble$_0$
        Else
            MRUDataOut$_{7-0}$ = baseByte
        EndIf


        # Setup MRUMode
        allowTrans = IsRAM $\lor$ IsRead $\lor$ (IsWrite $\land$ writeEn) $\lor$ IsErase
        If (doingTrans)
            MRUMode$_2$ = IsInfBlock
            MRUMode$_1$ = IsErase $\lor$ IsTestReg$_1$
            MRUMode$_0$  =  IsDeviceErase  $\lor$  ($\neg$IsWrite  $\land$  $\neg$IsPageErase)  $\lor$
        IsTestReg$_0$
            MRUNewTrans = startOfSubTrans $\land$ allowTrans $\land$
                        ($\neg$IsInvalidRAM $\lor$ MRUTestWE $\lor$ IsDeviceErase)
        Else
            MRUMode$_{2-0}$ = 001 # read (safe)
```

```
        MRUNewTrans = 0
    EndIf


    # Generate the effective nybble read from flash (this may not be
    used).
    # When there is a shadowFault (non-erased memory and invalid
    shadows) we consider
    # it a bad read when an 8-bit read, or when writeMask$_0$ is 0.
    # Note: we always substitute the upper nybble of WriteMask for the
    non-valid data,
    # but only flag a read error if WriteMask$_0$ is also 1. When the
    data is erased,
    # we return 0 regardless of WriteMask$_0$.
    finishedTrans = doingTrans ∧ MAURdy
    finishedFlashSubTrans = finishedTrans ∧ IsFlash ∧ ¬IsErase
    isWrittenFlash = (FlashData$_{7-0}$ ≠ 11111111)    # flash is erased to
    all 1s
    If (isWrittenFlash ∧ ((FlashData$_{7,5,3,1}$ ⊕ FlashData$_{6,4,2,0}$) ≠ 1111))
        inNybble$_{3-0}$ = WriteMask$_{7-4}$
        badRead  =  finishedFlashSubTrans  ∧  IsRead  ∧  (Is8Bit  ∨
    ¬WriteMask$_0$) ∧ doShadows
    Else
        inNybble$_{3,2,1,0}$ = (theMask$_{3,2,1,0}$ ⊕ FlashData$_{6,4,2,0}$) ∧ isWrittenFlash
        badRead = 0
    EndIf


    # Present the resultant data to the outside world
    MaskTheData = IsInvalidRAM ∨ badRead ∨ (badUntilRestart ∧ ¬IsRAM)
    NoData = IsErase ∨ IsWrite ∨ ¬doingTrans
    If (NoData ∨ MaskTheData)
        MRUData$_0$ = IsInvalidRAM ∧ illChip
        MRUData$_{4-1}$ = retryCount ∧ (IsInvalidRAM ∧ Adr$_2$)    # mask   all   4
    count bits
        MRUData$_{31-5}$ = 0 # also ensures a read that is bad returns 0
    ElseIf (IsRAM)
```

```
        MRUData₃₁₋₂₄ = SelectByte[RAMData, (Adr₁₋₀ ∨ Is32Bit|Is32Bit)]    #
    mux: 8 from 32
        MRUData₂₃₋₀ = RAMData₂₃₋₀  # lsbs remain unchanged from RAM
    ElseIf (doShadows)
        MRUData₃₁₋₂₈ = inNybble
        MRUData₂₇₋₀ = buff₂₇₋₀
    Else
        MRUData₃₁₋₂₄ = FlashData
        MRUData₂₃₋₀ = buff₂₇₋₄
    EndIf


    # Shift in the data for the good reads - either 4 or 8 bits
    (writes = don't care)
    If (finishedFlashSubTrans ∧ ¬badRead)
        buff₃₋₀ ← buff₇₋₄ # shift right 4 bits
        If (doShadows)
            buff₂₃₋₄ ← buff₂₇₋₈                                    #  shift  right  4
    bits
            buff₂₇₋₂₄ ← inNybble
        Else
            buff₁₉₋₄ ← buff₂₇₋₁₂  # shift right 8 bits, buff₃₋₀ is don't care
            buff₂₇₋₂₀ ← FlashData
        EndIf
    EndIf


    # Determine whether or not we need a new sub-transaction. We only
    need one if:
    # * there hasn't been a transition to IdleMode during this
    transaction
    # * we're doing 8 bit reads that are shadowed
    # * we're doing 32 bit reads and we've done less than 4 or 8 (sh
    vs non-sh)
    # * we got a bad read from flash and we need to retry the read
    (jic was a glitch)
    moreAdrsToGo = (¬toXfer₀ ∧ ((Is8Bit ∧ doShadows) ∨ Is32Bit)) ∨
                   (¬toXfer₁ ∧ Is32Bit) ∨ (¬toXfer₂ ∧ Is32Bit ∧ doShadows)
    needToRetryRead = badRead ∧ (¬retryStarted ∨ (retryCount ≠ 1111))
```

```
extraTrans_in  =  finishedFlashSubTrans  ∧  (moreAdrsToGo  ∨
needToRetryRead)
                    ∧ okForTrans
nextToXfer  ←  toXfer  +  (finishedFlashSubTrans  ∧  (IsWrite  ∨
¬needToRetryRead))


# generate our rdy signal and state values for next cycle
MRURdy = ¬doingTrans ∨ (doingTrans ∧ MAURdy ∧ ¬extraTrans_in)
extraTrans ← extraTrans_in
activeTrans ← ¬MRURdy   # all complete only when MRURdy is set


# Take account of bad reads
triedEnough = badRead ∧ retryStarted ∧ (retryCount = 1111)
If (MAURdy)
   If (IsTestModeWE ∧ (Adr$_{5-2}$ = 0000))   # capture  writes  to  local
regs
      illChip ← 0
      retryCount ← 0
   Else
      illChip ← illChip ∨ triedEnough
      If (badRead)
         retryCount ← (retryCount ∧ retryStarted) + 1 #  AND  all  4
bits
         retryStarted ← 1
      Else
         retryStarted ← 0# clear  flag  so  will  be  ok  for  the  next
read
      EndIf
   EndIf
EndIf


# Ensure that we won't have problems restarting a program
If (MRURdy ∧ ¬okForTrans) # note  MRURdy  (may  not  be  running  a
transaction!)
   shadowsOff,   hiFlashAdr,   infoBlockSel,   restartPending,
badUntilRestart ← 0
```

```
        Else
            badUntilRestart ← badUntilRestart ∨ triedEnough
            If (doingTrans ∧ ¬continueRequest)
                restartPending ← 1    # record for later use
            EndIf
            If (IsTestModeWE ∧ Adr₂) # the other writes are taken care of by
        the MAU
                shadowsOff ← ReqDataOut₀
                hiFlashAdr ← ReqDataOut₁
                infoBlockSel ← ReqDataOut₂
            EndIf
        EndIf
```

19 Memory Access Unit

The Memory Access Unit (MAU) takes memory access control signals and turns them into RAM accesses and flash access strobed signals with appropriate duration.

A new transaction is given by MRUNewTrans. The address to be read from or written to is on MRUAdr, which is a nybble-based address. The MRUAdr (13-bits) is used as-is for Flash addressing. When MRURAMSel = 1, then the RAM address (RAMAdr) is taken from bits 9-3 of MRUAdr. The data to be written is on MRUData.

The return value MAURdy is set when the MAU is capable of receiving a new transaction the following cycle. Thus MAURdy will be 1 during the final cycle of a flash or ram access, and should be 1 when the MAU is idle. MAURdy should only be 0 during startup or when a transaction has yet to finish.

When MRURAMSel = 1, the access is to RAM, and MRUMode has the following interpretation:

Table 384. Interpretation of MRUMode[10] for RAM accesses

bits	action
xx0	doWrite
xx1	doRead

When MRURAMSel = 0, the access is to flash. If MRUTestWE = 0, then the access is to regular flash memory, as given by MRUMode:

Table 385. Interpretation of MRUMode for regular flash accesses[11]

---

[10] $MRUMode_{2-1}$ is ignored for RAM accesses

[11] $MRUMode_2$ can be directly interpreted by the MAU as the IFREN signal required for embedded flash block SFC008_08B9_HE

bits1-0	action when MRUMode$_2$=0	action when MRUMode$_2$=1
00	doWrite (main memory)	doWrite (info block)
01	doRead (main memory)	doRead (info block)
10	doErasePage (main memory)	doErasePage (info block)
11	doEraseDevice (main memory)	doEraseDevice (both blocks)

If MRUTestWE is 1, then MRUMode$_2$ will also be 0, and the access is to a flash test register, as given by MRUMode:

Table 386. Interpretation of MRUMode for flash test register write accesses

bits[12]	action
xx1	If (MRUData$_3$ = 0), tie the flash IP test register to its reset state If (MRUData$_3$ = 1), take the flash IP test register out of reset state, and write MRUData$_{8-4}$ to the 5-bit flash test register within the flash IP (SFC008_08B9_HE)
x1x	Write MRUData$_{28-9}$ to the internal 20-bit alternate-counter-source register flashTime, and MRUData$_{29}$ to the corresponding 1-bit test register timerSel.

19.1   IMPLEMENTATION

The MAU consists of logic that calculates MAURdy, and additional logic that produces the various strobed signals according to the TSMC Flash memory SFC0008_08B9_HE; refer to this datasheet [4] for detailed timing diagrams. Both main memory and information blocks can be accessed in the Flash. The Flash test modes are also supported as described in [5] and general application information is given in [6].

The MAU can be considered to be a RAM control block and a flash control block, with appropriate action selected by MRURAMSel. For all modes except read, the Flash requires wait states (which are implemented with a single counter) during which it is possible to access the RAM. Only 1 transaction may be pending while waiting for the wait states to expire. Multiple bytes may be written to Flash without exiting the write mode.

---

[12]   MRUMode$_2$ will always be 0 when MRUTestWE = 1.

The MAU ensures that only valid control sequences meeting the timing requirements of the Flash memory are provided. A write time-out is included which ensures the Flash cannot be left in write mode indefinitely; this is used when the Flash is programmed via the IO Unit to ensure the X address does not change while in write mode. Otherwise, other units should ensure that when writing bytes to Flash, the X address does not change. The X address is held constant by the MAU during write and page erase modes to protect the Flash. If an X address change is detected by the MAU during a Flash write sequence, it will exit write mode allowing the X address to change and re-enter write mode. Thus, the data will still be written to Flash but it will take longer.

When either the Flash or RAM is not being used, the MAU sets the control signals to put the particular memory type into standby to minimise power consumption.

The MAU assumes no new transactions can start while one is in progress and all inputs must remain constant until MAU is ready.

19.2   FLASH TEST MODE

MAU also enables the Flash test mode register to be programmed which allows various production tests to be carried out. If MRUTestWE = 1, transactions are directed towards the test mode register. Most of the tests use the same control sequences that are used for normal operation except that one time value needs to be changed. This is provided by the flashTime register that can be written to by the CPU allowing the timer to be set to a range of values up to more than 1 second. A special control sequence is generated when the test mode register is set to 0x1E and is initiated by writing to the Flash.

Note that on reset, timeSel and flashTime are both cleared to 0. The 5-bit flash test register within the TSMC flash IP is also reset by setting TMR =1. When MRUTestWE = 1, any open write sequence is closed even if the write is not to the 5-bit flash test register within the TSMC flash IP.

19.3   FLASH POWER FAILURE PROTECTION

Power could fail at any time; the most serious consequence would be if this occurred during writing to the Flash and data became corrupted in another location to that being written to. The MAU will protect the Flash by switching off the charge pump (high voltage supply used for programming and erasing) as soon as the power starts to fail. After a time delay of about 5µs (programmable), to allow the discharge of the charge pump, the QA chip will be reset whether or not the power supply recovers.

19.4   FLASH ACCESS STATE MACHINE

19.5   INTERFACE

Table 387. MAU interface description

Signal name	I/O	Description
Clk	In	System clock.

1012

RstL	In	System reset (active low).
MAURAMEn	In	Flag indicating whether the external user needs access to the RAM at a gross level (e.g. the CPU is active and therefore may want RAM access). 1 = wants access available, 0 = don't want.
MRUNewTrans	In	Flag indicating MRU wishes to start a new transaction. May only be asserted (= 1) when MAURdy = 1. All inputs below must be held constant until MAU is ready.
MRURAMSel	In	1 = RAM, 0 = Flash.
MRUMode2-0	In	Type of transaction to be performed.
MRUAdr12-0	In	Memory address from the MRU.
MRUDataOut31-0	In	Data used to control and set test modes and timing.
MRUTestWE	In	Flag indicating test mode transactions.
PwrFailing	In	Flag indicating possible power failure in progress.
MAURdy	Out	The MAU is ready when MAURdy = 1. It is always set for RAM transactions and held low during Flash wait states.
RAMOutEn	Out	0 = enable the RAM to read or write this cycle (i.e. active low) 1 = disable the RAM this cycle (saves power, memory is intact)
RAMWE	Out	RAM write when RAMWE = 0 (Artisan Synchronous SRAM).
MemClk	Out	Inverted system clock to the RAM (required to meet timing).
FlashCtrl8-0	Out	Control signals to the Flash. IFREN = information block enable, not used always = 0 XE = X address enable YE = Y address enable SE = sense amplifier enable (read only) OE = output enable (read only), hi-Z when OE = 0 PROG = program (write bytes) NVSTR = enables all write and erase modes ERASE = page erase mode

1013

		MAS1 = mass erase mode
TMR	Out	TMR = Register reset for test mode
RAMAdr6-0	Out	RAM address in the range 0 to 95.
FlashAdr12-0	Out	Flash address, full range.
MAURstOutL	Out	Activates the global reset, RstL.

19.6 CALCULATION OF TIMER VALUES

Set and calculate timer initialisation values based on Flash data sheet values, clock period and clock range.

```
         # Note: Flash data sheet gives minimum timings
5        # Delays greater than 1 clock cycle


         clock_per     = 100     # ns


         Flash_Tnvs    = 7500    # ns
10       Flash_Tnvh    = 7500    # ns
         Flash_Tnvh1   = 150     # us
         Flash_Tpgs    = 15                        # us
         Flash_Tpgh    = 100     # ns
         Flash_Tprog   = 30      # us
15       Flash_Tads    = 100     # ns
         Flash_Tadh    = 30      # us   # Byte write timeout
         Flash_Trcv    = 1500    # ns
         Flash_Thv     = 6       # ms   # Not currently used
         Flash_Terase  = 30      # ms
20       Flash_Tme     = 300     # ms


         # Derive maximum counts (-1 since state machine is synchronous)
         FLASH_NVS     = Flash_Tnvs/clock_per - 1
         FLASH_NVH     = Flash_Tnvh/clock_per - 1
25       FLASH_NVH1    = Flash_Tnvh1*1000/clock_per - 1
         FLASH_PGS     = Flash_Tpgs*1000/clock_per - 1
         FLASH_PGH     = Flash_Tpgh/clock_per - 1
         FLASH_PROG    = Flash_Tprog*1000/clock_per - 1
         FLASH_ADS     = Flash_Tads/clock_per - 1
30       FLASH_ADH     = Flash_Tadh*1000/clock_per - 1
         FLASH_ADH_AND_WRITE_PGH = FLASH_ADH + FLASH_PGH + 1   # note is +1
         FLASH_RCV     = Flash_Trcv/clock_per - 1
```

1014

```
FLASH_HV    = Flash_Thv*1000000/clock_per - 1
FLASH_ERASE = Flash_Terase*1000000/clock_per - 1
FLASH_ME    = Flash_Tme*1000000/clock_per - 1


count_size = 24  # Number of bits in timer counter (newCount)
determined by Tme
```

19.7 DEFAULTS

Defaults to use when no action is specified.

```
FlashTransPendingSet = 0
FlashTransPendingReset = 0
TMRSet = 0
TMRRst = 0
STLESet = 0
STLERst = 0
TestTimeEn = 0
IFREN = FlashXadr₇
XE = 0
YE = 0
SE = 0
OE = 0
PROG = 0
NVSTR = 0
ERASE = 0
MAS1 = 0
MAURstOutL = 1


If (accessCount ≠ 0)
    newCount = accessCount - 1  # decrement unless instructed
otherwise
    Else
    newCount = 0
    EndIf
```

19.8 RESET

Initialise state and counter registers.

```
# asynchronous reset (active low)
    state ← idle
    accessCount ← 1
```

```
           countZ ← 0

           XadrReg ← 0

           FlashTransPending ← 0

           TestTime ← 0

5          TMR ← 1

           STLEFlag ← 0
```

19.9   STATE MACHINE

The state machine generates sequences of timed waveforms to control the operation of the Flash memory.

```
10         idle

           FlashTransPendingReset = 1

           If (somethingToDo) # Flash starting conditions

              If (MRUTestWE)

                 nextState = TM0

15            Else

                 Switch (MRUModeint)

                 Case doWrite:

                    nextState = writeNVS

                    newCount = FLASH_NVS

20               Case doRead:

                    YE = 1

                    SE = 1

                    OE = 1

                    XE = 1

25                  nextState = idle

                 Case doErasePage:

                    nextState = pageErase

                    newCount = FLASH_NVS

                 Case doEraseDevice:

30                  nextState = massErase

                    newCount = FLASH_NVS

                 EndSwitch

              EndIf

           EndIf
```

35   19.9.1   Flash page erase

The following pseducocode illustrates the Flash page erase sequence.

```
        pageErase
           ERASE = 1
           XE = 1
           If (¬PwrFailing)
  5            If (countZ)
                  newCount = FLASH_ERASE
                  nextState = pageEraseERASE
              EndIf
           Else
 10           newCount = TestTime$_{19-0}$
              nextState = Help1
           EndIf


        pageEraseERASE
 15        ERASE = 1
           NVSTR = 1
           XE = 1
           If (¬PwrFailing)
              If (countZ)
 20               newCount = FLASH_NVH
                  nextState = pageEraseNVH
              EndIf
           Else
              newCount = TestTime$_{19-0}$
 25           nextState = Help1
           EndIf


        pageEraseNVH
           NVSTR = 1
 30        XE = 1
           If (¬PwrFailing)
              If (countZ)
                  newCount = FLASH_RCV
                  nextState = RCVPM
 35           EndIf
           Else
              newCount = TestTime$_{19-0}$
```

1017

```
                    nextState = Help1
                    EndIf
                                                              .

             RCVPM
  5             If (countZ)
                   nextState = idle # exit
                EndIf
19.9.2    Flash mass erase
                The following pseducocode illustrates the Flash mass erase sequence.
 10      massErase
             MAS1 = 1
             ERASE = 1
             XE = 1
             If (countZ)
 15             If (¬TestTime20)
                   newCount = FLASH_ME
                Else
                   newCount = TestTime19-0 | 0000
                EndIf
 20             nextState = massEraseME
             EndIf


         massEraseME
             MAS1 = 1
 25          ERASE = 1
             NVSTR = 1
             XE = 1
             If (countZ)
                newCount = FLASH_NVH1
 30             nextState = massEraseNVH1
             EndIf


         massEraseNVH1
             MAS1 = 1
 35          NVSTR = 1
             XE = 1
             If (countZ)
```

1018

```
                          newCount = FLASH_RCV
                          nextState = RCVPM
                     EndIf
```

19.9.3   Flash byte write

The following pseducocode illustrates the Flash byte write sequence.

```
writeNVS
    PROG = 1
    XE = 1
    If (¬PwrFailing)
        If (countZ)
            If (¬STLEFlag)
                newCount = FLASH_PGS
                nextState = writePGS
            Else
                newCount = TestTime19-0 | 0000
                nextState = STLE0
            EndIf
        EndIf
    Else
        newCount = TestTime19-0
        nextState = Help1
    EndIf


writePGS
    PROG = 1
    NVSTR = 1
    XE = 1
    If (¬PwrFailing)
        If (countZ)
            newCount = FLASH_ADS
            nextState = writeADS
        EndIf
    Else
        newCount = TestTime19-0
        nextState = Help1
    EndIf
```

```
          writeADS # Add Tads to Tpgs
            PROG = 1
            NVSTR = 1
            XE = 1
5           FlashTransPendingReset = 1
            If (¬PwrFailing)
              If (countZ)
                If (¬TestTime20)
                  newCount = FLASH_PROG
10              Else
                  newCount = TestTime19-0 | 0000
                EndIf
                nextState = writePROG
              EndIf
15          Else
              newCount = TestTime19-0
              nextState = Help1
            EndIf


20        writePROG
            PROG = 1
            NVSTR = 1
            YE = 1
            XE = 1
25          If (¬PwrFailing)
              If (countZ)
                newCount = FLASH_ADH_AND_WRITE_PGH
                nextState = writeADH
              EndIf
30          Else
              newCount = TestTime19-0
              nextState = Help2
            EndIf


35        writeADH
            PROG = 1
            NVSTR = 1
```

```
        XE = 1
        FlashTransPendingSet = somethingToDo
        If (¬PwrFailing)
          If (¬FlashNewTrans)
            If (countZ)-- Gracefull exit after timeout
              newCount = FLASH_NVH
              nextState =writeNVH
            EndIf
          Else # -- Do something as there is a new transaction
            If ((MRUModeint = doWrite) ∧ (¬XadrCh))
              newCount = FLASH_ADS -- Write another byte
              nextState =writeADS
            Else
              newCount = FLASH_NVH -- Exit as new trans is not Flash
write
              nextState =writeNVH
            EndIf
          EndIf
        Else
          newCount = TestTime₁₉₋₀
          nextState =Help1
        EndIf


      writeNVH
        NVSTR = 1
        XE = 1
        FlashTransPendingSet = somethingToDo
        If (¬PwrFailing)
          If (countZ)
            newCount = FLASH_RCV
            nextState =RCV
          EndIf
        Else
          newCount = TestTime₁₉₋₀
          nextState =Help1
        EndIf
```

```
RCV          # wait til we're allowed to do another transaction
   FlashTransPendingSet = somethingToDo
   If (countZ)
      nextState = idle
   EndIf
```

19.9.4 Test Mode sequence

The following pseducocode illustrates the test mode sequence.

```
TM0 # Needed this due to delay on TMR
   IFREN = 0
   nextState = idle # default
   If ( MRUModeint₁)
      TestTimeEn = 1
   EndIf
   If (MRUModeint₀)
      If (¬MRUDataOut₃)
         TMRSet = 1
         STLERst = 1 # Reset flag as leaving test mode
      Else
         If (MRUDataOut₈₋₄ = 11110)
            STLESet = 1
         Else
            STLERst = 1
         EndIf
         TMRRst = 1
         nextState = TM1 # Will get priority
      EndIf
   EndIf


TM1
   IFREN = 0
   nextState = TM2


TM2
   NVSTR = 1
   SE = 1
   IFREN = 0
```

```
            nextState = TM3


        TM3
            NVSTR = 1
5           SE = 1
            MAS1 = MRUDataOut$_4$
            IFREN = MRUDataOut$_5$
            XE = MRUDataOut$_6$
            YE = MRUDataOut$_7$
10          ERASE = MRUDataOut$_8$
            TMRSet = 1
            nextState = TM4


        TM4
15          NVSTR = 1
            SE = 1
            MAS1 = MRUDataOut$_4$
            IFREN = MRUDataOut$_5$
            XE = MRUDataOut$_6$
20          YE = MRUDataOut$_7$
            ERASE = MRUDataOut$_8$
            TMRRst = 1
            nextState = TM5


25      TM5
            NVSTR = 1
            SE = 1
            MAS1 = MRUDataOut$_4$
            IFREN = MRUDataOut$_5$
30          XE = MRUDataOut$_6$
            YE = MRUDataOut$_7$
            ERASE = MRUDataOut$_8$
            nextState = TM6


35      TM6
            NVSTR = 1
            SE = 1
```

```
                    nextState = idle
19.9.5    Reverse tunneling and thin oxide leak test
          The following pseducocode shows the reverse tunneling and thin oxide leak test
          sequence.
STLE0
   XE = 1
   PROG = 1
   NVSTR = 1
   If (countZ)
      newCount = FLASH_NVH
      nextState = STLE1
   EndIf


STLE1
   XE = 1
   NVSTR = 1
   If (countZ)
      newCount = FLASH_RCV
      nextState = STLE2
   EndIf


STLE2
   If (countZ)
      nextState = idle
   EndIf
19.9.6    Emergency instructions
          The following pseducocode shows the states used for emergency situations such
          as when power is failing.
Help1 # MAURdy -> 0 to hold MAU inputs constant, if not too late
   XE = 1
   If (countZ)
      nextState = Goodbye
   EndIf


Help2 # MAURdy -> 0 to hold MAU inputs constant, if not too late
   XE = 1
   YE = 1
```

```
       If (countZ)
          nextState = Goodbye
       EndIf


5      Goodbye
          XE = 1 # Prevents Flash timing violation
          MAURstOutL = 0  # Reset whole chip whether power fails
                      # nothing else to do or recovers
19.10  CONCURRENT LOGIC
10     accessCount ← newCount # update accessCount every cycle
       countZ ← (newCount = 0)


       XadrReg ← FlashXAdr # store the previous X address
       state ← nextState
15
       If (FlashTransPendingReset)
          FlashTransPending ← 0 # Reset flag (has priority)
       Else
          If (FlashTransPendingSet)
20           FlashTransPending ← 1 # Set flag
          EndIf
       EndIf


       If (TestTimeEn)
25        TestTime ← MRUDataOut29-9
       EndIf


       If (TMRSet) -- SRFF for TMR
          TMR ← 1
30     Else
          If (TMRRst)
             TMR ← 0
          EndIf
       EndIf
35
       If (STLERst) -- SRFF for STLE tests
```

```
            STLEFlag ← 0
        Else
          If (STLESet)
             STLEFlag ← 1
          EndIf
        EndIf


        FlashNewTrans = MRUNewTrans ∧ (¬MRURAMSel)
        RAMNewTrans = MRUNewTrans ∧ MRURAMSel
        somethingToDo = FlashTransPending ∨ FlashNewTrans
        quickCmd = (MRUModeint = doRead) ∧ ¬MRUTestWE
        FlashRdy = ((state = idle) ∧ (¬somethingToDo ∨ quickCmd))
           ∨ (((state = writeADH)
           ∨ (state = writeNVH)
           ∨ (state = writeRCV)) ∧ (¬FlashTransPendingSet))
           ∨ ((state = TM0) ∧ (nextState = idle))
           ∨ (state = TM6)


        If (MRURamSel)
           MAURdy = 1 # Always ready for RAM
        Else
           MAURdy = FlashRdy
        EndIf


        IandX = MRUMode₂ | MRUAdr₁₂₋₆
        FlashXAdr = IandX When ((¬XE) ∨ (SE ∧ OE)) Else XadrReg
        FlashAdr = FlashXAdr | MRUAdr₅₋₀ # Merge X and Y addresses
        XadrCh  = 1  When  ((XadrReg  /=  IandX)  ∧  XE  ∧(¬SE)  ∧(¬OE)
        ∧ FlashNewTrans) Else 0
        # Xadr change


        MRUModeint = MRUMode₁₋₀ # Backwards compatability
        RAMAdr = MRUAdr₉₋₃   # maximum address = 95, responsibility of
        MRU for valid adr
        RAMWE = MRUModeint₀
        RAMOutEn = ¬RAMNewTrans # turn off RAM if not using it
```

```
            STLEFlag ← 0
        Else
          If (STLESet)
             STLEFlag ← 1
          EndIf
        EndIf


        FlashNewTrans = MRUNewTrans ∧ (¬MRURAMSel)
        RAMNewTrans = MRUNewTrans ∧ MRURAMSel
        somethingToDo = FlashTransPending ∨ FlashNewTrans
        quickCmd = (MRUModeint = doRead) ∧ ¬MRUTestWE
        FlashRdy = ((state = idle) ∧ (¬somethingToDo ∨ quickCmd))
           ∨ (((state = writeADH)
           ∨ (state = writeNVH)
           ∨ (state = writeRCV)) ∧ (¬FlashTransPendingSet))
           ∨ ((state = TM0) ∧ (nextState = idle))
           ∨ (state = TM6)


        If (MRURamSel)
           MAURdy = 1 # Always ready for RAM
        Else
           MAURdy = FlashRdy
        EndIf


        IandX = MRUMode_2 | MRUAdr_{12-6}
        FlashXAdr = IandX When ((¬XE) ∨ (SE ∧ OE)) Else XadrReg
        FlashAdr = FlashXAdr | MRUAdr_{5-0} # Merge X and Y addresses
        XadrCh  = 1  When  ((XadrReg  /=  IandX)  ∧  XE  ∧(¬SE)  ∧(¬OE)
        ∧ FlashNewTrans) Else 0
        # Xadr change


        MRUModeint = MRUMode_{1-0} # Backwards compatability
        RAMAdr = MRUAdr_{9-3}   # maximum address = 95, responsibility of
        MRU for valid adr
        RAMWE = MRUModeint_0
        RAMOutEn = ¬RAMNewTrans # turn off RAM if not using it
```

```
FlashCtrl(0)  = IFREN
FlashCtrl(1)  = XE
FlashCtrl(2)  = YE
FlashCtrl(3)  = SE
FlashCtrl(4)  = OE
FlashCtrl(5)  = PROG
FlashCtrl(6)  = NVSTR
FlashCtrl(7)  = ERASE
FlashCtrl(8)  = MAS1


MemClk        = ¬Clk # Memory clock
```

20 Analogue unit

This section specifies the mandatory blocks of Section 11.1 on page 965 in a way which allows some freedom in the detailed implementation.

Circuits need to operate over the temperature range -40°C to +125°C.

The unit provides power on reset, protection of the Flash memory against erroneous writes during power down (in conjunction with the MAU) and the system clock SysClk.

20.1 VOLTAGE BUDGET

The table below shows the key thresholds for V_{DD} which define the requirements for power on reset and normal operation.

Table 388. V_{DD} limits

VDD parameter	Description	Voltage
VDDFTmax	Flash test maximum	3.6[13]
VDDFTtyp	Flash test typical	3.3
VDDFTmin	Flash test minimum	3.0
VDDmax	Normal operation maximum (typ + 10%)	2.75[14]
VDDtyp	Normal operation typical	2.5
VDDmin	Normal operation minimum (typ - 5%)	2.375
VDDPORmax	Power on reset maximum	2.0[15]

---

[13] The voltage VDDFT may only be applied for the times specified in the TSMC Flash memory test document.

[14] Voltage regulators used to derive VDD will typically have symmetric tolerance lim its

[15] The minimum allowable voltage for Flash memory operation.

20.2   VOLTAGE REFERENCE

This circuit generates a stable voltage that is approximately independent of PVT (process, voltage, temperature) and will typically be implemented as a bandgap. Usually, a startup circuit is required to avoid the stable V_{bg} = 0 condition. The design should aim to minimise the additional voltage above V_{bg} required for the circuit to operate. An additional output, BGOn, will be provided and asserted when the bandgap has started and indicates to other blocks that the output voltage is stable and may be used.

Table 389. Bandgap target performance

Parameter	Conditions	Min	Typ	Max	Units
Vbg[16]	typical	1.2	1.23	1.26	V
IDD	typical		50		µA
Vstart	worst case	1.6			V
Iout				10	nA
Vtemp			+0.1		mV/oC

20.3   POWER DETECTION UNIT

Only under voltage detection will be described and is required to provide two outputs:

*   underL controls the power on reset; and
*   PwrFailing indicates possible failure of the power supply.

Both signals are derived by comparing scaled versions of V_{DD} against the reference voltage V_{bg}.

20.3.1 V_{DD} monotonicity

The rising and falling edges of V_{DD} (from the external power supply) shall be monotonic in order to guarantee correct operation of power on reset and power failing detection. Random noise may be present but should have a peak to peak amplitude of less than the hysteresis of the comparators used for detection in the PDU.

20.3.2   Under Voltage Detection Unit

The underL signal generates the global reset to the logic which should be de-asserted when the supply voltage is high enough for the logic and analogue circuits to operate. Since the logic reset is asynchronous, it is not necessary to ensure the clock is active before releasing the reset or to include any delay.

The QA chip logic will start immediately the power on reset is released so this should only be done when the conditions of supply voltage and clock frequency are within limits for the correct operation of the logic.

---

[16]   Over PVT, not including offsets

The power on reset signal shall not be triggered by narrow spikes (<100ns) on the power supply. Some immunity should be provided to power supply glitches although since the QA chip may be under attack, any reset delay should be kept short. The unit should not be triggered by logic dynamic current spikes resulting in short voltage spikes due to bond wire and package inductance.

5 On the rising edge of V_{DD}, the maximum threshold for de-asserting the signal shall be when $V_{DD} > V_{DDmin}$. On the falling edge of V_{DD}, the minimum threshold for asserting the signal shall be $V_{DD} < V_{DDPORmax}$.

The reset signal must be held low long enough (T_{pwmin}) to ensure all flip-flops are reset. The standard cell data sheet [7] gives a figure of 0.73ns for the minimum width of the reset pulse for all

10 flip-flop types.

2 bits of trimming ($trim_{1-0}$) will be provided to take up all of the error in the bandgap voltage. This will only affect the assertion of the reset during power down since the power on default setting must be used during power up.

Although the reference voltage cannot be directly measured, it is compared against V_{DD} in the PDU.

15 The state of the power on reset signal can be inferred by trying to communicate through the serial bus with the chip. By polling the chip and slowly increasing V_{DD}, a point will be reached where the power on reset is released allowing the serial bus to operate; this voltage should be recorded. As V_{DD} is lowered, it will cross the threshold which asserts the reset signal. The power on default is set to the lowest voltage that can be trimmed (which gives the maximum hysterisis). This voltage

20 should be recorded (or it may be sufficient to estimate it from the reset release voltage recorded above). V_{DD} is then increased above the reset release threshold and the PDU trim adjusted to the setting the closest to $V_{DDPORmax}$. V_{DD} should then be lowered and the threshold at which the reset is re-asserted confirmed.

Table 390. Power on reset target performance

25

Parameter	Conditions	Min	Typ	Max	Units
Vthrup	T = 27oC	2.0		2.375	V
Vthrdn	T = 27oC	2.0		2.1	V
Vhystmin			16		mV
IDD			5		µA
Tspike			100		ns
Vminr			0.5		V
Tpwmin		1			ns

Power on reset behaviour

The signal PwrFailing will be used to protect the Flash memory by turning off the charge pump during a write or page erase if the supply voltage drops below a certain threshold. The charge pump is

expected to take about 5us to discharge. The PwrFailing signal shall be protected against narrow spikes (< 100ns) on the power supply.

The nominal threshold for asserting the signal needs to be in the range $V_{PORmax} < V_{DDPFtyp} < V_{DDmin}$ so is chosen to be asserted when $V_{DD} < V_{DDPFtyp} = V_{DDPORmax} + 200mV$. This infers a V_{DD} slew rate limitation which must be < 200mV/5us to ensure enough time to detect that power is failing before the supply drops too low and the reset is activated. This requirement must be met in the application by provision of adequate supply decoupling or other means to control the rate of descent of V_{DD}.

Table 391. Power failing detection target performance

Parameter	Conditions	Min	Typ	Max	Units
Vthr	T = 27oC	2.1	2.2	2.3	V[17]
Vhyst			16		mV
IDD			5		µA
Tspike			100		ns
Vminr			0.5		V

2 bits of trimming ($trim_{1-0}$) will be provided to take up all of the error in the bandgap voltage.

20.4   RING OSCILLATOR

SysClk is required to be in the range 7 - 14 MHz throughout the lifetime of the circuit provided V_{DD} is maintained within the range $V_{DDMIN} < V_{DD} < V_{DDMAX}$. The 2:1 range is derived from the programming time requirements of the TSMC Flash memory. If this range is exceeded, the useful lifetime of the Flash may be reduced.

The first version of the QA chip, without physical protection, does not require the addition of random jitter to the clock. However, it is recommended that the ring oscillator be designed in such a way as to allow for the addition of jitter later on with minimal modification. In this way, the un-trimmed centre frequency would not be expected to change.

The initial frequency error must be reduced to remain within the range 10MHz / 1.41 to 10MHz × 1.41 allowing for variation in:

•   voltage

•   temperature

•   ageing

•   added jitter

•   errors in frequency measurement and setting accuracy

The range budget must be partitioned between these variables.

Figure 411._ Ring oscillator block diagram

---

[17]   These limits are after trimming and include an allowance for VDD ramping.

The above arrangement allows the oscillator centre frequency to be trimmed since the bias current of the ring oscillator is controlled by the DAC. SysClk is derived by dividing the oscillator frequency by 5 which makes the oscillator smaller and allows the duty cycle of the clock to be better controlled.

20.4.1   DAC (programmable current source)

Using V_{bg}, this block sources a current that can be programmed by the Trim signal. 6 of the available 8 trim bits will be used ($trim_{7-2}$) giving a clock adjustment resolution of about 250kHz. The range of current should be such that the ring oscillator frequency can be adjusted over a 4 to 1 range.

Table 392. Programmable current source target performance

Parameter	Conditions	Min	Typ	Max	Units
Iout	Trim7-2 = 0		5		µA
	Trim7-2 = 32		12.5		
	Trim7-2 = 63		20		
Vrefin			1.23		V
Rout	Trim7-2 = 63	2.5			MΩ

20.4.2   Ring oscillator circuit

Table 393. Ring oscillator target performance

Parameter	Conditions	Min	Typ	Max	Units
Fosc[18]		7	10	14	MHz
IDD			10		µA
KI			1		MHz/µA
KVDD			+200		KHz/V
KT			+30		KHz/oC
Vstart		1.5			V

K_I = control sensitivity, K_{VDD} = V_{DD} sensitivity, K_T = temperature sensitivity

With the figures above, K_{VDD} will give rise to a maximum variation of ±50kHz and K_T to ±1.8MHz over the specified range of V_{DD} and temperature.

20.4.3   Div5

The ring oscillator will be prescaled by 5 to obtain the nominal 10MHz clock. An asynchronous design may be used to save power. Several divided clock duty cycles are obtainable, eg 4:1, 3:2

---

[18]   Accounting for division by 5

etc. To ease timing requirements for the standard cell logic block, the following clock will be generated; most flip-flops will operate on the rising edge of the clock allowing negative edge clocking to meet memory timing.

Table 394. Div5 target performance

Parameter	Conditions	Min	Typ	Max	Units
Fmax	Vdd = 1.5V	100			MHz
IDD			10		µA

20.5   POWER ON RESET

This block combines the overL (omitted from the current version), underL and MAURstOutL signals to provide the global reset. MAURstOutL is delayed by one clock cycle to ensure a reset generated when this signal is asserted has at least this duration since the reset deasserts the signal itself. It should be noted that the register, with active low reset RN, is the only one in the QA chip not connected to RstL.

[4]   TSMC, Oct 1, 2000, *SFC0008_08B9_HE*, 8K × 8 Embedded Flash Memory Specification, Rev 0.1.

[5]   TSMC (design service division), Sep 10, 2001, *0.25um Embedded Flash Test Mode User Guide*, V0.3.

[6]   TSMC (EmbFlash product marketing), Oct 19, 2001, *0.25um Application Note*, V2.2.

[7]   Artisan Components, Jan 99, Process Perfect Library Databook 2.5-Volt Standard Cells, Rev1.0.

OTHER APPLCATIONS FOR PROTOCOLS AND QA CHIPS

1   Introduction

In its preferred form, the QA chip [1] is a programmable 32 bit microprocessor with security features (8,000 gates, 3k bits of RAM and 8kbytes of flash memory for program and non-volatile data

5   storage). It is manufactured in a 0.25 um CMOS process.

Physically, the chip is mounted in a 5 pin SOT23 plastic package and communicates with external circuitry via a two pin serial bus.

10   The QA chip was designed to for authenticating consumable usage and performance upgrades in printers and associated hardware.

Because of its core functionality and programmability the QA chip can also be used in applications that differ significantly from its original one. This document seeks to identify some of those areas.

15

3   Applications Overview

Applications include:

- Regular EEPROM
- Secure EEPROM

20   - General purpose MPU with security features
- Security coprocessor for microprocessor system
- Security coprocessor for PC (with optional USB connection)
- Resource dispenser - secure, web based transfer of a variable quantity from "source" to "sink"
- ID tag

25   - Security pass inside offices
- Set top box security
- Car key
- Car Petrol
- Car manufacturer "genuine parts" detection, where the car requires genuine (or authorised)

30   parts to function.
- Aeroplane control on motor-control servos to allow secure external control on an aircraft in a hijack situation.
- Security device for controlling access to and copying of audio, video, and data (eg, preventing unauthorized downloading of music to a device).

35

4 Exemplary Application Descriptions

4.1 Car Petrol

Using mechanisms and protocols similar to those described in relation to ink refills, refilling of petrol can be controlled. An example of a commercial relationship this allows is selling a car at a discounted rate, but requiring that the car be refilled at designated service stations. Similarly, prevention of unauthorized servicing can be achieved.

4.2 Car Keys

4.2.1 BASIC ADVANTAGES OVER PHYSICAL KEYS

- Keys and locks can be easily programmed & configured for use
- Can only be duplicated/reprogrammed by an authorised individual
- The same key can be used for physical entry/exit and remote (radio-based) entry/exit
- Inbuilt security features

4.2.2 SINGLE KEY FOR MULTIPLE VEHICLES

Useful when a family has more than one car.

- Can be programmed so any keys fits any car.
- Fewer number of duplicate keys.
- Misplacing a key for a particular car - any key for any other car can be used as oppose to duplicate of the same key.

4.2.3 MULTIPLE KEYS FOR A SINGLE VEHICLE

4.2.3.1 Same company car being driven by multiple drivers

- Mileage can be logged per driver e.g. for accounting purposes.
- Key permissions can be different per driver (e.g. boot/trunk access may be disabled)

4.2.3.2 Same family car being driven by children and parents

- Time/date restrictions can be applied to (e.g. children's) keys
- Speeds above a specified limit (and duration of that speed) can be logged for auditing purposes (may be less dangerous than actually enforcing a speed limit)

4.2.4 NO PROBLEM IF KEY LOST

Can easily:

- make a new key the same as lost one (existing copies of key will still function)
- reprogram the locks on car (and reprogram all non-lost keys to match) so the lost key will no longer function

4.2.5 NO PROBLEM IF KEY LEFT IN CAR

- Easy to create a one-time-use open-door-only key via roadside assistance based on secret password information, driver's license etc (prevents having to break into the car)

4.2.6 CAR RENTALS

- Key can have an expiration date (e.g. some period past the rental end-date)

4.2.7 SINGLE PHYSICAL KEY FOR ALL LOCKS IN CAR

A single physical key can open all locks (door, immobiliser, boot/trunk, glovebox etc.).

```c
#define INTERP 1

#include "srm015.c"

#if INTERP
module stitch_module {
#endif

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "ldata.h"
#include "func.h"

#define STITCH_MAR 0.3
#define GRID_STEP 0.025
#define BIN_SIZE 128.0

    typedef enum { LEFT, RIGHT, DONOTKNOW } halves;
    static int bin_size = 0;
    static LCoord stitch_mar = 0;
    static LCoord grid_step = 0;
#define otherSide(s)  ((s==DONOTKNOW)?DONOTKNOW:(s==LEFT)?RIGHT:LEFT)

#define AND 1
#define OR 0
#define NOT 1
#define ACTUAL 0

    int do_Poly = 1;
    int do_Metal1 = 1;
    int do_Metal2 = 1;
    int do_Metal3 = 1;
    int do_Metal4 = 1;
    int do_Contact = 1;
    int do_Glass = 1;
    int do_Via1 = 1;
    int do_Via2 = 1;
    int do_Via3 = 1;
    int do_NDiffusion = 1;
    int do_PDiffusion = 1;
    int do_NTie = 1;
    int do_PTie = 1;
    int do_NWell = 1;
    int do_PWell = 1;
    int do_PSelect = 1;
    int do_NSelect = 1;

    void SetupSplitLayers()
        {

            LDialogItem items1 [ 11 ] =
                {
                    { "Poly", "1"},
                    { "Metal1", "1"},
                    { "Metal2", "1"},
                    { "Metal3", "1"},
```

```
                    { "Metal4", "1"},
                    { "Contact", "1"},
                    { "Glass", "1"},
                    { "Via1", "1"},
                    { "Via2", "1"},
                    { "Via3", "1"},
                    { "more ...", "1"}
            };
        LDialogItem items2 [ 8 ] =
            {
                    { "N Diffusion", "1"},
                    { "P Diffusion", "1"},
                    { "NTie", "1"},
                    { "PTie", "1"},
                    { "N Well", "1"},
                    { "P Well", "1"},
                    { "P+ Select", "1"},
                    { "N+ Select", "1"}
            };

        strcpy(items1[0].value, do_Poly ? "1" : "0");
        strcpy(items1[1].value, do_Metal1 ? "1" : "0");
        strcpy(items1[2].value, do_Metal2 ? "1" : "0");
        strcpy(items1[3].value, do_Metal3 ? "1" : "0");
        strcpy(items1[4].value, do_Metal4 ? "1" : "0");
        strcpy(items1[5].value, do_Contact ? "1" : "0");
        strcpy(items1[6].value, do_Glass ? "1" : "0");
        strcpy(items1[7].value, do_Via1 ? "1" : "0");
        strcpy(items1[8].value, do_Via2 ? "1" : "0");
        strcpy(items1[9].value, do_Via3 ? "1" : "0");
        strcpy(items2[0].value, do_NDiffusion ? "1" : "0");
        strcpy(items2[1].value, do_PDiffusion ? "1" : "0");
        strcpy(items2[2].value, do_NTie ? "1" : "0");
        strcpy(items2[3].value, do_PTie ? "1" : "0");
        strcpy(items2[4].value, do_NWell ? "1" : "0");
        strcpy(items2[5].value, do_PWell ? "1" : "0");
        strcpy(items2[6].value, do_PSelect ? "1" : "0");
        strcpy(items2[7].value, do_NSelect ? "1" : "0");

        if ( LDialog_MultiLineInputBox( "Split layer to process", items1,
11 ) ) // failes with more than 15 objects
            {
                /* A OK was hit by the user, so get the property value from
the Dialog_Items buffer*/
                do_Poly = atoi(items1[0].value);
                do_Metal1 = atoi(items1[1].value);
                do_Metal2 = atoi(items1[2].value);
                do_Metal3 = atoi(items1[3].value);
                do_Metal4 = atoi(items1[4].value);
                do_Contact = atoi(items1[5].value);
                do_Glass = atoi(items1[6].value);
                do_Via1 = atoi(items1[7].value);
                do_Via2 = atoi(items1[8].value);
                do_Via3 = atoi(items1[9].value);
                if ( atoi(items1[10].value)
                    && LDialog_MultiLineInputBox( "Split layer to
process", items2, 8 ) )
                    {
                        do_NDiffusion = atoi(items2[0].value);
                        do_PDiffusion = atoi(items2[1].value);
                        do_NTie = atoi(items2[2].value);
```

```
                              do_PTie = atoi(items2[3].value);
                              do_NWell = atoi(items2[4].value);
                              do_PWell = atoi(items2[5].value);
                              do_PSelect = atoi(items2[6].value);
                              do_NSelect = atoi(items2[7].value);
                      }
              }
      }


    halves whichSideOfWire(LObject object, LPoint p)
        {
              LVertex v;
              LPoint s, e;

              if ( (v = LObject_GetVertexList(object)) == NULL ) return
DONOTKNOW;
              s = LVertex_GetPoint(v);
              for ( v=LVertex_GetNext(v); v != NULL; v=LVertex_GetNext(v))
              {
                  e = LVertex_GetPoint(v);
                  if ( s.y != e.y )
                  {
                      if( (s.y <= p.y && p.y <= e.y) ||
                          (s.y >= p.y && p.y >= e.y) )
                      {
                          assert( s.x == e.x, "wire is not orothanal");
                          if ( p.x <= s.x ) return LEFT;
                          return RIGHT;
                      }
                  }
                  s = e;
              }
              return DONOTKNOW;
        }


    long getSelfSpacingRule(LFile File, char *lname)
        {
              LDrcRule rule;
              LDesignRuleParam *p, drcp;

              if( (rule = LDrcRule_Find(File, LSPACING, lname, NULL)) == NULL )
              {
                  if( (rule = LDrcRule_Find(File, LSPACING, lname, "")) == NULL
)
                  {
                      if( (rule = LDrcRule_Find(File, LSPACING, lname, lname))
== NULL )
                      {
                          for( rule = LDrcRule_GetList(File); rule != NULL;
rule = LDrcRule_GetNext(rule))
                              {
                                  assert( p = LDrcRule_GetParameters( rule, &drcp),
"no drc parameters");
                                  if ( p->rule_type == LSPACING && strcmp(lname, p-
>layer1) == 0 )
                                  {
                                      if ( p->layer2 == NULL || strcmp(p-
>layer2,"") == 0 || strcmp(p->layer2, lname) == 0 )
                                      {
                                          assert(0,"spacing found by search");
```

```
                                              assert(0, p->layer2 == NULL ? "(null)" :
p->layer2);
                                              break;
                                      }
                              }
                      }
                  }
              }
          assert( rule, "no drc found parameters");
          assert( p = LDrcRule_GetParameters( rule, &drcp), "no drc
parameters");

          return p->distance;
      }

   void deleteTmpLayer(LFile File, LLayer layer)
       {
          LSelection_DeselectAll();
          LSelection_AddAllObjectsOnLayer(layer);
          LSelection_Clear();
          LLayer_Delete(File,layer);
       }

   LStatus generateLayer3Op( LFile File, LCell cell, LLayer copy,
                          int    not_l1, LLayer  l1, int    grow_l1,
                          int    op1,
                          int    not_l2, LLayer  l2, int    grow_l2,
                          int    op2,
                          int    not_l3, LLayer  l3, int    grow_l3)
   {
          LDerivedLayerParam dpt;
          LDerivedLayerParam *d;
          char copy_name[64];
          char l1_name[64];
          char l2_name[64];
          char l3_name[64];
          if (bin_size == 0 ) bin_size = (int)LFile_LocUtoIntU(File,
BIN_SIZE);


          d = &dpt;
          d->enable_evaluation = 1;
          d->name = LLayer_GetName(copy, copy_name, 64);
          d->layer1_not_op = not_l1;
          d->src_layer1 = LLayer_GetName(l1, l1_name, 64);
          d->layer1_grow_amount = grow_l1;
          d->layer1_bool_layer2 = op1;
          d->layer2_not_op = not_l2;
          if ( l2 != NULL )
              d->src_layer2 = LLayer_GetName(l2, l2_name, 64);
          else
              d->src_layer2 = "";
          d->layer2_grow_amount = grow_l2;
          d->layer2_bool_layer3 = op2;
          d->layer3_not_op = not_l3;
          if ( l3 != NULL )
              d->src_layer3 = LLayer_GetName(l3, l3_name, 64);
          else
              d->src_layer3 = "";
          d->layer3_grow_amount = grow_l3;
```

```
                IsOK( LLayer_SetDerivedParameters(File, copy, d), "Set derived
op3");
                return ( LCell_GenerateLayersEx00(cell, bin_size, copy, LFALSE,
LFALSE) );
        }

#define copyLayer( File, cell, copy, orig, grow) generateLayer3Op( File,
cell, copy, \
                                        ACTUAL, orig, grow, AND, NOT, NULL, 0, AND,
NOT, NULL, 0)

#define generateLayer2op( File, cell, copy, not_l1, l1, grow_l1, op, not_l2,
l2, grow_l2) \
        generateLayer3Op( File, cell, copy, not_l1, l1, grow_l1, op, not_l2,
l2, grow_l2, AND, NOT, NULL, 0)


    LRect selectionBbox()
        {
                LSelection sel;
                LObject obj;
                LRect bb;
                LRect rect;

                sel = LSelection_GetList();
                obj = LSelection_GetObject(sel);
                bb = LObject_GetMbb(obj);
                for( sel = LSelection_GetNext(sel); sel != NULL;
                        sel = LSelection_GetNext(sel) )
                {
                    obj = LSelection_GetObject(sel);
                    rect = LObject_GetMbb(obj);
                    if ( rect.y0 < bb.y0 ) bb.y0 = rect.y0;
                    if ( rect.y1 > bb.y1 ) bb.y1 = rect.y1;
                    if ( rect.x0 < bb.x0 ) bb.x0 = rect.x0;
                    if ( rect.x1 > bb.x1 ) bb.x1 = rect.x1;
                }
                return bb;
        }

    LLayer createLayer( LFile File, LLayer after, char *new_name)
        {
                if( LLayer_New(File, after, new_name) == LStatusOK )
                    return LLayer_GetNext(after);
                else
                    return GetLLayer(File, new_name);
        }

    void copyLayerToTop(LFile File, LCell cell, LLayer l)
        {
                LLayer tmp;

                VISIBLE(l);
                assert( tmp = createLayer(File, l, "tmp_for_copy"), "create tmp
layer");
                IsOK( copyLayer(File, cell, tmp, l, 0), "copy layer");
                LSelection_DeselectAll();
                IsOK(LSelection_AddAllObjectsOnLayer(tmp),"find tmp");
                IsOK(LSelection_ChangeLayer(tmp, l), "change tmp to l");
                LLayer_Delete(File,tmp);
        }
```

```
    /*
     * create left and right halves zones either side of the cut_line,
     * optionally generate ilayer, the work_layer in a zone around  the
cut_line
     * The later exist after the cut_line, left_area, right_area [, ilayer]
     */
    LStatus create_halfs( LFile File, LCell cell, LLayer cut_line, LLayer
work_layer )
        {
            LLayer wframe, both_sides;
            LLayer cut_copy, left_area, right_area, ilayer;
            LSelection sel;
            LObject obj;
            LRect cbb;
            LRect bb;
            LRect rect;
            LRect frame;

            cut_copy = createLayer(File, cut_line, "cut_copy");

            /* create a copy of the cut line in the current level
             * cut_copy = cut
             * only way to detetrmine that a cut line exist in side the cell
             * at some level
             */
            IsOK( copyLayer(File, cell, cut_copy, cut_line, 0), "copy cut
line");

            LSelection_DeselectAll();
            if ( LSelection_AddAllObjectsOnLayer(cut_copy) != LStatusOK )
            {
            // LDialog_MsgBox("Error:  No cut line found.");
                deleteTmpLayer(File,cut_copy);
                return LBadCell;
            }

            right_area = createLayer(File, cut_copy, "right_area");
            left_area = createLayer(File, cut_copy, "left_area");
            both_sides = createLayer(File, cut_copy, "both_sides");
            wframe = createLayer(File, cut_copy, "wframe");

            cbb = LCell_GetMbb(cell);

            bb = selectionBbox();

            frame.x0 = cbb.x0;
            frame.x1 = cbb.x1;
            frame.y0 = bb.y0;
            frame.y1 = bb.y1;
            // wframe = create a bounding box around the cut line to the edge
of the cell
            LBox_New(cell, wframe, frame.x0, frame.y0, frame.x1, frame.y1);

            // cut wframe in half about the cut line
            //  both_sides = !cut & wframe;
            IsOK( generateLayer2op(File, cell, both_sides, NOT, cut_copy, 0,
AND, ACTUAL, wframe, 0),"cut out cut_line");

            LSelection_DeselectAll();
            LSelection_AddAllObjectsOnLayer(both_sides);
```

```
                LSelection_Merge();

                sel = LSelection_GetList();
                obj = LSelection_GetObject(sel);
                rect = LObject_GetMbb(obj);
                if ( frame.x0 != rect.x0 )
                {
                     sel = LSelection_GetNext(sel);
                     obj = LSelection_GetObject(sel);
                }
                LSelection_RemoveObject(obj);
                LSelection_ChangeLayer(both_sides,left_area);

                LSelection_AddAllObjectsOnLayer(both_sides);
                LSelection_ChangeLayer(both_sides,right_area);

                if ( work_layer != NULL )
                {
                     ilayer = createLayer(File, right_area, "ilayer");
                     // create a smaller bounding box hopefully to keep the work
load down
                     LSelection_DeselectAll();
                     LSelection_AddAllObjectsOnLayer(wframe);
                     LSelection_Clear();
                     rect.x0 = bb.x0 - stitch_mar;
                     rect.x1 = bb.x1 + stitch_mar;
                     rect.y0 = bb.y0;
                     rect.y1 = bb.y1;
                     LBox_New(cell, wframe, rect.x0, rect.y0, rect.x1, rect.y1);
                     // ilayer = layer of interest inside this bonding box
                     // ilayer = wframe & layer

                     IsOK( generateLayer2op(File, cell, ilayer, ACTUAL, wframe, 0,
AND, ACTUAL, work_layer, 0), "layer of interest");
                }

                deleteTmpLayer(File,both_sides);
                deleteTmpLayer(File,wframe);
                deleteTmpLayer(File,cut_copy);
                return LStatusOK;
        }

    void createSideMaterial(LFile File, LCell cell,
                            LLayer t5, LLayer t6, LLayer t7, LLayer t8,
                            LLayer side, LLayer not_side, LLayer side_area,
LLayer maskSide,
                            long spacing)
        {
                LSelection_DeselectAll();
                LSelection_AddAllObjectsOnLayer(t6);
                LSelection_AddAllObjectsOnLayer(t7);
                LSelection_Clear();


                // grow a little to recover & remote unwanted right side material
                // t6 = grow(t5,.15) & !not_right
                // IsOK( generateLayer2op(File, cell, t6, ACTUAL, t5,
stitch_mar/2, AND, NOT, not_side, 0), "regrow and remove unwanted right");
                if ( not_side != NULL )
                {
```

```
                IsOK( generateLayer2op(File, cell, t6, ACTUAL, t5, 0, AND,
NOT, not_side, 0), "remove unwanted right");

                // t7 = grow(t6,.275) or right_area
                IsOK( generateLayer2op(File, cell, t7, ACTUAL, t6, spacing,
OR, ACTUAL, side_area, spacing), "merge right");

            }
            else
            {
                // t7 = grow(t5,.275) or right_area
                IsOK( generateLayer2op(File, cell, t7, ACTUAL, t5, spacing,
OR, ACTUAL, side_area, spacing), "merge right");
            }
            // t8 = grow(t7,-.275) and ! right_area
            IsOK( generateLayer2op(File, cell, t8, ACTUAL, t7, -spacing, AND,
NOT, side_area, 0), "remove right area");

            LSelection_DeselectAll();
            LSelection_AddAllObjectsOnLayer(t8);
            //LSelection_Merge();
            LSelection_ChangeLayer(t8,side);
            LSelection_DeselectAll();
            if ( not_side == NULL )
            {
                LSelection_AddAllObjectsOnLayer(side_area);
                LSelection_ChangeLayer(side_area,maskSide);
            }
            else
            {
                LSelection_AddAllObjectsOnLayer(t6);
                LSelection_Clear();

                IsOK( generateLayer2op(File, cell, t6, ACTUAL, side_area, 0,
AND, NOT, not_side, 0),
                        "remove not_side area");
                LSelection_AddAllObjectsOnLayer(t6);
                LSelection_ChangeLayer(t6,maskSide);
            }
        }
    }

    void createSplitLayer(LFile File, LCell cell, LLayer work_layer)
        {
            char layer_name[64];
            char cut_name[64];
            char left_name[64];
            char right_name[64];
            char maskL_name[64];
            char maskR_name[64];
            char no_left_name[64];
            char no_right_name[64];
            LLayer maskR;
            LLayer maskL;
            LLayer cut_line, left, right;
            LLayer not_left, not_right;
            LLayer ilayer, t4, t5, t6;
            LLayer left_area, right_area, t7, t8;
            LSelection sel;
            LObject obj;
            long spacing;
```

```
            step_no = 0;

            if( stitch_mar == 0 ) stitch_mar = LFile_LocUtoIntU(File,
STITCH_MAR);
            if( grid_step == 0 )  grid_step = LFile_LocUtoIntU(File,
GRID_STEP);
            LLayer_GetName(work_layer, layer_name, 64);

            spacing = getSelfSpacingRule(File, layer_name)/ 2;

            strcpy(cut_name, layer_name);
            strncat(cut_name, " cut line", 64);
            strcpy(left_name, layer_name);
            strncat(left_name, " left", 64);
            strcpy(right_name, layer_name);
            strncat(right_name, " right", 64);
            strcpy(no_left_name, layer_name);
            strncat(no_left_name, " not left", 64);
            strcpy(no_right_name, layer_name);
            strncat(no_right_name, " not right", 64);
            strcpy(maskL_name, layer_name);
            strncat(maskL_name, " maskL", 64);
            strcpy(maskR_name, layer_name);
            strncat(maskR_name, " maskR", 64);

            if ( (cut_line = GetLLayer(File, cut_name)) == NULL ) return;

            if ( (left = GetLLayer(File, left_name)) == NULL ) return;
            if ( (right = GetLLayer(File, right_name)) == NULL ) return;
            if ( (maskL = GetLLayer(File, maskL_name)) == NULL ) return;
            if ( (maskR = GetLLayer(File, maskR_name)) == NULL ) return;
            // LFile_Save(File);
            VISIBLE(work_layer);
            VISIBLE(cut_line);
            VISIBLE(left);
            VISIBLE(right);
            VISIBLE(maskL);
            VISIBLE(maskR);
            if ( (not_left = LLayer_Find(File, no_left_name)) != NULL )
VISIBLE(not_left);
            if ( (not_right = LLayer_Find(File, no_right_name)) != NULL )
VISIBLE(not_right);

            SetMsg(layer_name,"");
            LSelection_DeselectAll();
            if ( LSelection_AddAllObjectsOnLayer(left) == LStatusOK )
LSelection_Clear();
            if ( LSelection_AddAllObjectsOnLayer(right) == LStatusOK )
LSelection_Clear();
            if ( LSelection_AddAllObjectsOnLayer(maskL) == LStatusOK )
LSelection_Clear();
            if ( LSelection_AddAllObjectsOnLayer(maskR) == LStatusOK )
LSelection_Clear();

            if ( create_halfs(File, cell, cut_line, work_layer) != LStatusOK
) return;

            left_area = LLayer_GetNext(cut_line);
            right_area = LLayer_GetNext(left_area);
            ilayer = LLayer_GetNext(right_area);
```

```
                t8 = createLayer( File, ilayer, "t8");
                t7 = createLayer( File, ilayer, "t7");
                t6 = createLayer( File, ilayer, "t6");
                t5 = createLayer( File, ilayer, "t5");
                t4 = createLayer( File, ilayer, "t4");

                // create interestion area with cut line
                // t4 = ilayer AND cut
                IsOK( generateLayer2op(File, cell, t4, ACTUAL, ilayer, 0, AND,
ACTUAL, cut_line, 0), "merge cut and ilayer");
                // t5 = grow (t4, stitch_mar);
                IsOK( copyLayer(File, cell, t5, t4, stitch_mar), "grow");


                createSideMaterial(File, cell, t5, t6, t7, t8, left, not_left,
left_area, maskL, spacing);
                createSideMaterial(File, cell, t5, t6, t7, t8, right, not_right,
right_area, maskR, spacing);

                deleteTmpLayer(File,t8);
                deleteTmpLayer(File,t7);
                deleteTmpLayer(File,t6);
                deleteTmpLayer(File,t5);
                deleteTmpLayer(File,t4);
                deleteTmpLayer(File,ilayer);
                deleteTmpLayer(File,left_area);
                deleteTmpLayer(File,right_area);
        }

    void createSplit(void)
        {
                LFile File;
                LLayer ll;

                LCell cell;
                if ( (cell = getCurrentCell(&File)) == NULL ) return;
                ll = GetSelectedLayer(cell);

                if ( ll != NULL ) createSplitLayer(File, cell, ll);
                resetUpi();
        }

    void divide_material( LCell cell, LObject cut_object, LLayer layer,
LLayer left, LLayer right)
        {
                LObject o;
                LRect bb;
                LPoint p;

                LSelection_DeselectAll();
                for ( o = LObject_GetList(cell, layer); o != NULL; o =
LObject_GetNext(o))
                {
                    bb = LObject_GetMbb(o);
                    p.x = (bb.x0 + bb.x1)/2;
                    p.y = (bb.y0 + bb.y1)/2;
                    if ( whichSideOfWire(cut_object, p) == LEFT )
                    {
                        LSelection_AddObject(o);
                    }
                }
```

```
                LSelection_ChangeLayer(layer, left);
                LSelection_DeselectAll();
                LSelection_AddAllObjectsOnLayer(layer);
                LSelection_ChangeLayer(layer, right);
                LSelection_DeselectAll();
        }

     void createDivideLayer(LFile File, LCell cell, LLayer work_layer)
        {
                char layer_name[64];
                char cut_name[64];
                char left_name[64];
                char right_name[64];
                char psuedo_name[64];
                LLayer left_area, right_area;
                LLayer cut_line, left, right;
                LLayer psuedo_layer;
                LObject cut_object;



                if ( work_layer == NULL ) return;
                LLayer_GetName(work_layer, layer_name, 64);

                strcpy(cut_name, layer_name);
                strncat(cut_name, " cut line", 64);
                strcpy(left_name, layer_name);
                strncat(left_name, " left", 64);
                strcpy(right_name, layer_name);
                strncat(right_name, " right", 64);
                strcpy(psuedo_name, "psuedo_");
                strncat(psuedo_name, layer_name, 64);


                if ( (cut_line = GetLLayer(File, cut_name)) == NULL ) return;

                if ( (left = GetLLayer(File, left_name)) == NULL ) return;
                if ( (right = GetLLayer(File, right_name)) == NULL ) return;
                VISIBLE(work_layer);
                VISIBLE(cut_line);
                VISIBLE(left);
                VISIBLE(right);
                if ( (psuedo_layer = LLayer_Find(File, psuedo_name)) != NULL )
VISIBLE(psuedo_layer);

                step_no = 0;

                LSelection_DeselectAll();
                LSelection_AddAllObjectsOnLayer(left);
                LSelection_AddAllObjectsOnLayer(right);
                LSelection_Clear();

                LSelection_DeselectAll();
                LSelection_AddAllObjectsOnLayer(cut_line);
                LSelection_Merge();
                cut_object = LObject_GetList(cell, cut_line);
                assert( LObject_GetNext(cut_object) == NULL, "more than one cut
area");

                divide_material(cell, cut_object, work_layer, left, right);
```

```
                 if ( psuedo_layer != NULL )
                 {
                     strcpy(left_name, psuedo_name);
                     strncat(left_name, " left", 64);
                     strcpy(right_name, psuedo_name);
                     strncat(right_name, " right", 64);
                     left = GetLLayer(File, left_name);
                     right = GetLLayer(File, right_name);
                     VISIBLE(left);
                     VISIBLE(right);
                     divide_material(cell, cut_object, psuedo_layer, left, right);
                 }

            }

        void createDivide(void)
            {
                 LFile File;
                 LLayer ll;

                 LCell cell;
                 if ( (cell = getCurrentCell(&File)) == NULL ) return;
                 ll = GetSelectedLayer(cell);

                 if ( ll != NULL ) createDivideLayer(File, cell, ll);
                 resetUpi();
            }

/*
 * creates 3 new temperary layers left, right, work,
 * returns pointer to the left, and the other will follow it
 * left is the area to the left of the cut line, including the cut line
 * right is the area to the right of the cut line, including the cut line
 * work is a tempary layer for other to use
 */
        LLayer createOverlapFields(LFile File, LCell cell, char *layer_name)
            {
                 char cut_name[64];
                 LLayer cut_line;
                 LLayer left_area, right_area;
                 LLayer t7;
                 LLayer left, right, work;


                 strcpy(cut_name, layer_name);
                 strncat(cut_name, " cut line", 64);

                 if ( (cut_line = GetLLayer(File, cut_name)) == NULL ) return
NULL;

                 VISIBLE(cut_line);


                 if ( create_halfs(File, cell, cut_line, NULL) != LStatusOK )
return;

                 left_area = LLayer_GetNext(cut_line);
                 right_area = LLayer_GetNext(left_area);
```

```
        ⌐ t7 = createLayer( File, right_area, "t7");
          work = createLayer( File, right_area, "work");
          right = createLayer( File, right_area, "right");
          left = createLayer( File, right_area, "left");


          generateLayer2op(File, cell, t7, ACTUAL, cut_line, 0, OR, ACTUAL,
right_area, 0);
          LSelection_DeselectAll();
          LSelection_AddAllObjectsOnLayer(t7);
          LSelection_ChangeLayer(t7,left);

          LSelection_DeselectAll();
          LSelection_AddAllObjectsOnLayer(t7);
          LSelection_Clear();

          generateLayer2op(File, cell, t7, ACTUAL, cut_line, 0, OR, ACTUAL,
left_area, 0);
          LSelection_DeselectAll();
          LSelection_AddAllObjectsOnLayer(t7);
          LSelection_ChangeLayer(t7,right);

          deleteTmpLayer(File,t7);
          if ( do_PWell )
          {
              LLayer pw_maskL, pw_maskR;
              pw_maskL = GetLLayer(File, "P Well maskL");
              pw_maskR = GetLLayer(File, "P Well maskR");
              LSelection_DeselectAll();
              LSelection_AddAllObjectsOnLayer(right_area);
              LSelection_ChangeLayer(right_area,pw_maskR);
              LSelection_DeselectAll();
              LSelection_AddAllObjectsOnLayer(left_area);
              LSelection_ChangeLayer(left_area,pw_maskL);
          }
          deleteTmpLayer(File,right_area);
          deleteTmpLayer(File,left_area);

          return ( left );
      }

   void createOverlapLayer(LFile File, LCell cell, char *layer_name, LLayer
left)
          {
          char cut_name[64];
          char left_name[64];
          char right_name[64];
          LLayer work_layer;
          LLayer work_left, work_right;
          LLayer work, right;
          LSelection sel;
          LObject obj;
          LRect bb;
          LRect rect;
          LRect frame;
          LCoord tmp;
          LDerivedLayerParam *d;

          strcpy(left_name, layer_name);
          strncat(left_name, " left", 64);
          strcpy(right_name, layer_name);
```

```
                strncat(right_name, " right", 64);

                if ( (work_layer = GetLLayer(File, layer_name)) == NULL ) return;

                if ( (work_left = GetLLayer(File, left_name)) == NULL ) return;
                if ( (work_right = GetLLayer(File, right_name)) == NULL ) return;
                VISIBLE(work_layer);
                VISIBLE(work_left);
                VISIBLE(work_right);

                LSelection_DeselectAll();
                LSelection_AddAllObjectsOnLayer(work_left);
                LSelection_AddAllObjectsOnLayer(work_right);
                LSelection_Clear();

                right = LLayer_GetNext(left);
                work = LLayer_GetNext(right);


                //  work = work_layer & left
                generateLayer2op(File, cell, work, ACTUAL, work_layer, 0, AND,
ACTUAL, left, 0);

                LSelection_DeselectAll();
                LSelection_AddAllObjectsOnLayer(work);
                LSelection_ChangeLayer(work,work_left);

                //  work = work_layer & right
                generateLayer2op(File, cell, work, ACTUAL, work_layer, 0, AND,
ACTUAL, right, 0);

                LSelection_DeselectAll();
                LSelection_AddAllObjectsOnLayer(work);
                LSelection_ChangeLayer(work,work_right);
        }


    void doOverlappingSplit(LFile File, LCell cell)
        {
                LLayer left, right, work;

                left = createOverlapFields(File, cell, "P Well");
                if ( left == NULL )
                {
                    return;
                }
                right = LLayer_GetNext(left);
                work  = LLayer_GetNext(right);
                if ( do_NDiffusion ) createOverlapLayer(File, cell, "N
Diffusion", left);
                if ( do_PDiffusion ) createOverlapLayer(File, cell, "P
Diffusion", left);
                if ( do_NTie )      createOverlapLayer(File, cell, "NTie", left);
                if ( do_PTie )      createOverlapLayer(File, cell, "PTie", left);
                if ( do_NWell )     createOverlapLayer(File, cell, "N Well",
left);
                if ( do_PWell )     createOverlapLayer(File, cell, "P Well",
left);
                if ( do_PSelect )   createOverlapLayer(File, cell, "P+ Select",
left);
```

```
                 if ( do_NSelect )  createOverlapLayer(File, cell, "N+ Select",
left);

                 deleteTmpLayer(File,work);
                 deleteTmpLayer(File,right);
                 deleteTmpLayer(File,left);
        }

    LCell completeSplit(LFile File, LCell cell)
        {
                 char cell_name[64];
                 char new_name[64];
                 LCell New;
                 LLayer l;

                 assert( LCell_GetName(cell, cell_name, 64) != NULL, "no cell
name");
                 strcpy(new_name, cell_name);
                 strncat(new_name, "_cut", 64 );
                 SetMsg(cell_name,new_name);
                 LCell_Copy(File, cell, File, new_name);
                 assert( (New = LCell_Find(File, new_name)) != NULL, "new cell
creation failed");
                 New = LCell_Flatten(New);
                 assert(New != NULL,"flatten side");
                 LCell_MakeVisibleNoRefresh(New);

                 if ( do_Poly )       createSplitLayer(File, New, GetLLayer(File,
"Poly"));
                 if ( do_Metal1 )    createSplitLayer(File, New, GetLLayer(File,
"Metal1"));
                 if ( do_Metal2 )    createSplitLayer(File, New, GetLLayer(File,
"Metal2"));
                 if ( do_Metal3 )    createSplitLayer(File, New, GetLLayer(File,
"Metal3"));
                 if ( do_Metal4 )    createSplitLayer(File, New, GetLLayer(File,
"Metal4"));
                 if ( do_Contact )   createDivideLayer(File, New, GetLLayer(File,
"Contact"));
                 if ( do_Via1 )       createDivideLayer(File, New, GetLLayer(File,
"Via1"));
                 if ( do_Via2 )       createDivideLayer(File, New, GetLLayer(File,
"Via2"));
                 if ( do_Via3 )       createDivideLayer(File, New, GetLLayer(File,
"Via3"));
                 if ( do_Glass )      createDivideLayer(File, New, GetLLayer(File,
"Glass"));
                 doOverlappingSplit(File, New);

                 LSelection_DeselectAll();
                 l = GetLLayer(File, "n+ implant");
                 VISIBLE(l);
                 LSelection_AddAllObjectsOnLayer(l);
                 LSelection_Clear();
                 l= GetLLayer(File, "p+ implant");
                 VISIBLE(l);
                 LSelection_AddAllObjectsOnLayer(l);
                 LSelection_Clear();
                 l= GetLLayer(File, "nwell");
                 VISIBLE(l);
                 LSelection_AddAllObjectsOnLayer(l);
```

```
            LSelection_Clear();

            return New;
        }


    void makeSideLayer2(LFile File, LCell Cell, char *l_name, halves Side,
                int copy, char *omask_ext)
        {
            char s_name[64];
            char o_name[64];
            char m_name[64];
            char c_name[64];
            LLayer Layer_mask;
            LLayer Layer_side;
            LLayer Layer_orig;
            LLayer Layer_other;
            LLayer Layer_cut;
            char cell_name[64];

            char *oside_ext = (Side == RIGHT)? " left" : " right";
            char *side_ext = (Side == LEFT)? " left" : " right";

            assert( LCell_GetName(Cell, cell_name, 64) != NULL, "no cell
name");

            SetMsg(cell_name,l_name);
            // LDialog_MsgBox(cell_name);

            strcpy(s_name, l_name);
            strncat(s_name, side_ext, 64);
            strcpy(o_name, l_name);
            strncat(o_name, oside_ext, 64);

            Layer_orig = GetLLayer(File,l_name);
            assert(Layer_orig != NULL,l_name);
            VISIBLE(Layer_orig);

            Layer_side = GetLLayer(File,s_name);
            assert(Layer_side != NULL,s_name);
            VISIBLE(Layer_side);

            Layer_other = GetLLayer(File, o_name);
            assert(Layer_other != NULL,o_name);
            VISIBLE(Layer_other);

            LSelection_DeselectAll();

            if ( omask_ext != NULL )
            {
                LRect r, c;
                LPoint p;
                LObject o;
                LLayer tmp;

                strcpy(m_name, l_name);
                strncat(m_name, omask_ext, 64);
                Layer_mask = GetLLayer(File, m_name);
                assert(Layer_mask != NULL,m_name);
                VISIBLE(Layer_mask);
```

```
                    strcpy(c_name, l_name);
                    strncat(c_name, " cut line", 64);
                    Layer_cut = GetLLayer(File, c_name);
                    assert(Layer_cut != NULL,c_name);
                    VISIBLE(Layer_cut);
                    IsOK(LSelection_AddAllObjectsOnLayer(Layer_cut),"get
cut_line");
                    r =  selectionBbox();
                    c = LCell_GetMbbAll(Cell);
                    if ( Side == LEFT  )
                        p.x = c.x0 = r.x1;
                    else
                        p.x = c.x1 = r.x0;
                    p.y = 0;
                    LSelection_DeselectAll();
                    LSelection_AddAllObjectsOnLayer(Layer_orig);
                    LSelection_SliceVertical(&p);

                    // tmp = createLayer(File, Layer_cut, "tmp_mask");

                    // generateLayer2op( File, Cell, tmp, ACTUAL, Layer_orig, 0,
AND, ACTUAL, Layer_mask, 0);
                    // LSelection_AddAllObjectsOnLayer(tmp);
                    // LSelection_ChangeLayer(tmp, Layer_side);

                    // LSelection_DeselectAll();
                    // for ( o = LObject_GetList(Cell, Layer_orig); o != NULL; o
= LObject_GetNext(o))
                    // {
                        // switch( LObject_GetShape(o) )
                        // {
                        // case LTorus:
                        // case LCircle:
                        // case LPie:
                            // LSelection_AddObject(o);
                            // break;
                        // }
                    // }
                    LSelection_RemoveAllObjectsInRect(&c);
                    LSelection_ChangeLayer(Layer_orig, Layer_side);
                    LSelection_DeselectAll();

                    LSelection_AddAllObjectsOnLayer(Layer_orig);
                    LSelection_AddAllObjectsOnLayer(Layer_mask);
                    LSelection_Clear();
                    // LLayer_Delete(File,tmp);
                }

                // delete the other side material
                LSelection_AddAllObjectsOnLayer(Layer_other);
                LSelection_Clear();

                if( copy  == 0 )
                {
                    // delete orig material and copy the
                    // side material to it

                    LSelection_AddAllObjectsOnLayer(Layer_orig);
                    LSelection_Clear();
                    LSelection_AddAllObjectsOnLayer(Layer_side);
                    LSelection_ChangeLayer(Layer_side, Layer_orig);
```

```
            }
            // else keep both

        }

    void makeSideCell2(LFile File, LCell Cell, halves Side)
        {
            char *omask_ext = (Side == RIGHT)? " maskL" : " maskR";
            LLayer l;


            LCell_MakeVisibleNoRefresh(Cell);
            if ( do_Poly ) makeSideLayer2(File, Cell, "Poly",
Side, 1, omask_ext);
            if ( do_Metal1 ) makeSideLayer2(File, Cell, "Metal1", Side, 1,
omask_ext);
            if ( do_Metal2 ) makeSideLayer2(File, Cell, "Metal2", Side, 1,
omask_ext);
            if ( do_Metal3 ) makeSideLayer2(File, Cell, "Metal3", Side, 1,
omask_ext);
            if ( do_Metal4 ) makeSideLayer2(File, Cell, "Metal4", Side, 1,
omask_ext);
            if ( do_Contact ) makeSideLayer2(File, Cell, "Contact",
Side, 0, NULL);
            if ( do_Via1 ) makeSideLayer2(File, Cell, "Via1",
Side, 0, NULL);
            if ( do_Via2 ) makeSideLayer2(File, Cell, "Via2",
Side, 0, NULL);
            if ( do_Via3 ) makeSideLayer2(File, Cell, "Via3",
Side, 0, NULL);
            if ( do_Glass ) makeSideLayer2(File, Cell, "Glass",
Side, 0, NULL);
            if ( do_Contact ) makeSideLayer2(File, Cell, "psuedo_Contact",
        Side, 0, NULL);
            if ( do_Via1 ) makeSideLayer2(File, Cell, "psuedo_Via1",
        Side, 0, NULL);
            if ( do_Via2 ) makeSideLayer2(File, Cell, "psuedo_Via2",
        Side, 0, NULL);
            if ( do_Via3 ) makeSideLayer2(File, Cell, "psuedo_Via3",
        Side, 0, NULL);
            if ( do_Glass ) makeSideLayer2(File, Cell, "psuedo_Glass",
        Side, 0, NULL);
            if ( do_NDiffusion ) makeSideLayer2(File,Cell, "N Diffusion",
        Side, 0, NULL);
            if ( do_PDiffusion ) makeSideLayer2(File,Cell, "P Diffusion",
        Side, 0, NULL);
            if ( do_NTie ) makeSideLayer2(File, Cell, "NTie",
Side, 0, NULL);
            if ( do_PTie ) makeSideLayer2(File, Cell, "PTie",
Side, 0, NULL);
            if ( do_NWell ) makeSideLayer2(File, Cell, "N Well", Side, 0,
NULL);
            if ( do_PWell ) makeSideLayer2(File, Cell, "P Well", Side, 0,
omask_ext);
            if ( do_PSelect ) makeSideLayer2(File, Cell, "P+ Select",
Side, 0, NULL);
            if ( do_NSelect ) makeSideLayer2(File, Cell, "N+ Select",
Side, 0, NULL);


        }
```

```
    void makeNewSideCell(LFile File, LCell Cell, halves  Side)
        {
            char cell_name[64];
            char new_name[64];
            LCell New;

            assert( LCell_GetName(Cell, cell_name, 64) != NULL, "no cell
name");
            strcpy(new_name, cell_name);
            strncat(new_name, (Side == LEFT) ? "_left" : "_right", 64);
            SetMsg(cell_name,new_name);
            LCell_Copy(File, Cell, File, new_name);
            assert( (New = LCell_Find(File, new_name)) != NULL, "new cell
creation failed");
            makeSideCell2(File,New,Side);
        }


    void doOverlapSplit(void)
        {
            LFile File;

            LCell cell;
            if ( (cell = getCurrentCell(&File)) == NULL ) return;

            doOverlappingSplit(File, cell);

            resetUpi();
        }



    void see_cut_layers( )
        {
            LFile File;

            LCell Cell_Draw;
            if ( (Cell_Draw = getCurrentCell(&File)) == NULL ) return;

            // cut on
            make_visible( GetLLayer(File, "cut start"),
                          GetLLayer(File, "cut end"),
                          LVisible, LIGNORE, File);

            resetUpi();
        }

    void splitAllInstances()
        {
            LFile File;
            LLayer l;
            LCell cell;
            LCell subcell;
            LInstance inst;
            int delete = 0;

            if ( (cell = getCurrentCell(&File)) == NULL ) return;

            for ( inst = LInstance_GetList(cell); inst != NULL; inst =
LInstance_GetNext(inst))
                {
                    subcell = LInstance_GetCell(inst);
                    createSidesDo(File, subcell);
```

```
                }
                LCell_MakeVisibleNoRefresh(cell);

                resetUpi();
        }


    char *cutname( char *orig, char *new, halves Side, int n)
        {
                char *ptr;

                strncpy(new, orig, n);
                strncat(new, (Side == DONOTKNOW) ? "_join" : (Side == LEFT) ?
"_cut_left" : "_cut_right", n);
                return( new );
        }


    /* copy cell to cell_cut_SIDE or cell_join, and replace all instances
with
     * subinst_cut_side if they exist
     */
    void createEnd(LFile File, LCell Cell, halves  Side)
        {
                char cell_name[128];
                char new_name[128];
                LCell New;
                LCell subcell;
                LCell newsub;
                LInstance inst;
                LInstance ninst;
                LTransform xform;
                LPoint repeat, delta;

                assert( LCell_GetName(Cell, cell_name, 128) != NULL, "no cell
name");
                cutname(cell_name, new_name, Side, 128);
                LCell_Copy(File, Cell, File, new_name);
                assert( (New = LCell_Find(File, new_name)) != NULL, "new cell
creation failed");

                for ( inst = LInstance_GetList(New); inst != NULL; inst = ninst )
                {
                    ninst = LInstance_GetNext(inst);
                    subcell = LInstance_GetCell(inst);
                    assert( LCell_GetName(subcell, cell_name, 128) != NULL, "no
(sub) cell name");
                    xform = LInstance_GetTransform(inst);
                    if ( xform.orientation == 0 )
                        cutname(cell_name, new_name, Side, 128);
                    else
                        cutname(cell_name, new_name, otherSide(Side), 128);
                    if( (newsub = LCell_Find(File, new_name)) != NULL )
                    {
                        if ( (newsub = LCell_Find(File, new_name)) == NULL )
                        {
                            LDialog_MsgBox("Error:  side subside does not
exists.");
                            LDialog_MsgBox(new_name);
                            return;
                        }

                        repeat = LInstance_GetRepeatCount(inst);
```

1055

```
                    delta = LInstance_GetDelta(inst);
                    if( LInstance_New(New,newsub,xform, repeat, delta) ==
NULL )
                    {
                        LDialog_MsgBox("Error:   side sub instance creation
failed.");
                        LDialog_MsgBox(new_name);
                    }
                    LInstance_Delete(New, inst);
                }
            }
        }


    mergeLayers(LFile File, LCell cell, LLayer work_layer, LLayer side, LLayer mask,
                LLayer t8, LLayer t9)
        {
            LObject o, n;
            LObject b;
            LRect r;

            generateLayer2op(File, cell, t8, ACTUAL, side, 0, AND, ACTUAL,
mask, 0);
            LSelection_DeselectAll();
            LSelection_AddAllObjectsOnLayer(t8);
            LSelection_ChangeLayer(t8, work_layer);

            if ( t9 != NULL )
            {
                for ( o = LObject_GetList(cell, side); o != NULL; o = n )
                {
                    n = LObject_GetNext(o);
                    switch( LObject_GetShape(o) )
                    {
                    case LTorus:
                    case LCircle:
                    case LPie:
                        r = LObject_GetMbb(o);
                        b = LBox_New(cell, t9, r.x0, r.y0, r.x1, r.y1);
                        generateLayer2op(File, cell, t8, ACTUAL, t9, 0, AND,
ACTUAL, mask, 0);
                        LSelection_DeselectAll();
                        if ( LSelection_AddAllObjectsOnLayer(t8) == LStatusOK
)
                        {
                            LSelection_Clear();
                            LObject_ChangeLayer(cell, o, work_layer );
                        }
                        break;
                    }
                }
            }
        }


    void mergeSides(LFile File, LCell cell, char *layer_name)
        {
            LLayer work_layer;
            char cut_name[64];
            char left_name[64];
            char right_name[64];
            char maskL_name[64];
```

```
            char maskR_name[64];
            char no_left_name[64];
            char no_right_name[64];
            LLayer maskR;
            LLayer maskL;
            LLayer cut_line, left, right;
            LLayer left_area, right_area, t8, t9;
            LSelection sel;
            LObject obj;

            if ( (work_layer = GetLLayer(File, layer_name)) == NULL ) return;

            strcpy(cut_name, layer_name);
            strncat(cut_name, " cut line", 64);
            strcpy(left_name, layer_name);
            strncat(left_name, " left", 64);
            strcpy(right_name, layer_name);
            strncat(right_name, " right", 64);
            strcpy(maskL_name, layer_name);
            strncat(maskL_name, " maskL", 64);
            strcpy(maskR_name, layer_name);
            strncat(maskR_name, " maskR", 64);

            if ( (cut_line = GetLLayer(File, cut_name)) == NULL ) return;
            if ( (left = GetLLayer(File, left_name)) == NULL ) return;
            if ( (right = GetLLayer(File, right_name)) == NULL ) return;
            if ( (maskL = GetLLayer(File, maskL_name)) == NULL ) return;
            if ( (maskR = GetLLayer(File, maskR_name)) == NULL ) return;

            VISIBLE(work_layer);
            VISIBLE(left);
            VISIBLE(right);
            VISIBLE(maskL);
            VISIBLE(maskR);

            t8 = createLayer( File, cut_line, "t8");
            t9 = createLayer( File, cut_line, "t9");
            mergeLayers(File, cell, work_layer, left, maskR, t8, t9);
            mergeLayers(File, cell, work_layer, right, maskL, t8, t9);
            mergeLayers(File, cell, work_layer, right, left, t8, NULL);

            LSelection_DeselectAll();
            LSelection_AddAllObjectsOnLayer(right);
            LSelection_AddAllObjectsOnLayer(left);
            LSelection_AddAllObjectsOnLayer(maskL);
            LSelection_AddAllObjectsOnLayer(maskR);
            LSelection_Clear();

            deleteTmpLayer(File,t8);
            deleteTmpLayer(File,t9);
        }

    copyPorts(LCell orig, LCell  New)
        {
            LPort port;
            char pname[50];
            LLayer l;
            LRect r;

            for( port = LPort_GetList(orig); port != NULL; port =
LPort_GetNext(port))
```

```
                {
                        LPort_GetText(port, pname, 50);
                        l = LPort_GetLayer(port);
                        r = LPort_GetRect(port);
                        assert(LPort_New(New, l, pname, r.x0, r.y0, r.x1, r.y1),"copy
port fail");
                }
        }

    void createJoin(LFile File, LCell cell)
        {
                char cell_name[128];
                char new_name[128];
                char sub_name[128];
                LCell New;
                LCell subcell;
                LInstance inst;
                LInstance ninst;
                LTransform xform;
                LPoint repeat, delta;

                assert( LCell_GetName(cell, cell_name, 128) != NULL, "no cell
name");

                strcpy(new_name, cell_name);
                strcat(new_name, "_join");

                cutname(cell_name, sub_name, LEFT, 128);
                assert( (subcell = LCell_Find(File, sub_name)) != NULL, "find
left side");

                LCell_Copy(File, subcell, File, new_name);
                assert( (New = LCell_Find(File, new_name)) != NULL, "new join
cell creation failed");

                xform.translation.x = 0;
                xform.translation.y = 0;
                xform.orientation = 0;
                xform.magnification.num = 1;
                xform.magnification.denom = 1;
                repeat.x = 1;
                repeat.y = 1;
                delta.x = 1;
                delta.y = 1;
                cutname(cell_name, new_name, RIGHT, 128);
                assert( (subcell = LCell_Find(File, new_name)) != NULL, "find
right side");
                LInstance_New(New,subcell,xform, repeat, delta);
                LCell_MakeVisibleNoRefresh(New);
                New = LCell_Flatten(New);
                if ( do_Poly   ) mergeSides(File, New, "Poly");
                if ( do_Metal1 ) mergeSides(File, New, "Metal1");
                if ( do_Metal2 ) mergeSides(File, New, "Metal2");
                if ( do_Metal3 ) mergeSides(File, New, "Metal3");
                if ( do_Metal4 ) mergeSides(File, New, "Metal4");
                if ( do_PWell  ) mergeSides(File, New, "P Well");

                copyPorts(cell, New);
        }

    void createSidesDo(LFile File, LCell cell)
        {
```

```
            LCell cut_cell;

            LCell_MakeVisibleNoRefresh(cell);
            cut_cell = completeSplit(File, cell);
            makeNewSideCell(File, cut_cell, LEFT);
            makeNewSideCell(File, cut_cell, RIGHT);
            createJoin(File, cell);
        }

    void createSides(void)
        {
            LFile File;

            LCell cell;
            LCell cut_cell;
            if ( (cell = getCurrentCell(&File)) == NULL ) return;

            createSidesDo(File, cell);

            resetUpi();
        }

    void createEnds(LFile File, LCell Cell)
        {
            createEnd(File, Cell, LEFT);
            createEnd(File, Cell, RIGHT);
            createEnd(File, Cell, DONOTKNOW);
        }

    void do_createEnds( )
        {
            LFile File;

            LCell cell;
            if ( (cell = getCurrentCell(&File)) == NULL ) return;

            createEnds(File, cell);

            resetUpi();
        }

    void copy_cut_lines( )
        {
            LFile File;

            LCell cell;

            LLayer ll, start, end;
            char lname[64];
            char *ptr;

            if ( (cell = getCurrentCell(&File)) == NULL ) return;

            start = GetLLayer(File, "cut start");
            end = GetLLayer(File, "cut end");
            for ( ll = LLayer_GetNext(start); ll != end; ll =
LLayer_GetNext(ll)) {
                    LLayer_GetName(ll, lname, 64);
                    ptr = lname + strlen(lname) - 9;
                    if ( strcmp( ptr," cut line") == 0 )
```

```
                {
                        copyLayerToTop(File, cell, ll);
                }
        }
        resetUpi();
}


void see_cut_lines( )
    {
        LFile File;

        LCell cell;

        LLayer ll, start, end;
        char lname[64];
        char *ptr;

        if ( (cell = getCurrentCell(&File)) == NULL ) return;

        start = GetLLayer(File, "cut start");
        end = GetLLayer(File, "cut end");
        for ( ll = LLayer_GetNext(start); ll != end; ll =
LLayer_GetNext(ll)) {
                LLayer_GetName(ll, lname, 64);
                ptr = lname + strlen(lname) - 9;
                if ( strcmp( ptr," cut line") == 0 )
                {
                        VISIBLE(ll);
                }
        }
        resetUpi();
    }


void createEndsAllInstances()
    {
        LFile File;
        LLayer l;
        LCell cell;
        LCell subcell;
        LInstance inst;
        int delete = 0;

        if ( (cell = getCurrentCell(&File)) == NULL ) return;

        for ( inst = LInstance_GetList(cell); inst != NULL; inst =
LInstance_GetNext(inst))
            {
                subcell = LInstance_GetCell(inst);
                createEnds(File, subcell);
            }
        LCell_MakeVisibleNoRefresh(cell);

        resetUpi();
    }

void createJoinDo(void)
    {
        LFile File;

        LCell cell;
```

1060

```
                LCell cut_cell;
                if ( (cell = getCurrentCell(&File)) == NULL ) return;

                createJoin(File, cell);

                resetUpi();
        }

    void createJoinAllInstances()
        {
                LFile File;
                LLayer l;
                LCell cell;
                LCell subcell;
                LInstance inst;
                int delete = 0;

                if ( (cell = getCurrentCell(&File)) == NULL ) return;

                for ( inst = LInstance_GetList(cell); inst != NULL; inst =
    LInstance_GetNext(inst))
                    {
                        subcell = LInstance_GetCell(inst);
                        createJoin(File, subcell);
                    }
                LCell_MakeVisibleNoRefresh(cell);

                resetUpi();
        }

    void doAll()
        {
                LFile File;
                LCell cell;
                LInstance inst;
                LObject o;
                LLayer l;
                if ( (cell = getCurrentCell(&File)) == NULL ) return;

                // cell = LCell_Find(File, "gen_gds_list");
                // LCell_MakeVisibleNoRefresh(cell);
                // genGDSforAllInstance();

                cell = LCell_Find(File, "stitch_list");
                LCell_MakeVisibleNoRefresh(cell);
                splitAllInstances();

                cell = LCell_Find(File, "end1_list");
                LCell_MakeVisibleNoRefresh(cell);
                createEndsAllInstances();

                cell = LCell_Find(File, "end2_list");
                LCell_MakeVisibleNoRefresh(cell);
                createEndsAllInstances();

                resetUpi();
        }

    void delete_joins()
        {
                LFile File;
```

```c
            LCell cell;
            LCell ncell;
            char *ptr;
            char cell_name[64];

            if ( (cell = getCurrentCell(&File)) == NULL ) return;

            for( cell = LCell_GetList(File); cell != NULL; cell = ncell )
            {
                ncell = LCell_GetNext(cell);
                assert( LCell_GetName(cell, cell_name, 64) != NULL, "no cell
name");
                ptr = cell_name + strlen(cell_name) - 5;
                if ( strcmp( ptr,"_join") == 0 )
                {
                    if( LDialog_YesNoBox(cell_name) )
                        LCell_Delete(cell);
                }
            }

            resetUpi();
        }

#if INTERP
    void stitch_macro_register( void )
#else
        int UPI_Entry_Point( void )
#endif
        {
            LMacro_Register("create Split", "createSplit");
            LMacro_Register("create Divide", "createDivide");
            LMacro_Register("do overlap split", "doOverlapSplit");
            LMacro_Register("create Sides", "createSides");
            // LMacro_Register("create Join", "createJoinDo");
            LMacro_Register("split all instances", "splitAllInstances");
            LMacro_Register("Set up Split", "SetupSplitLayers");
            LMacro_Register("See cut layers", "see_cut_layers");
            LMacro_Register("See cut lines", "see_cut_lines");
            LMacro_Register("Copy cut lines", "copy_cut_lines");
            LMacro_Register("create ends cells", "do_createEnds");
            LMacro_Register("create ends all instances",
"createEndsAllInstances");
            // LMacro_Register("create join all instances",
"createJoinAllInstances");
            LMacro_Register("do all", "doAll");
            LMacro_Register("delete _join cells", "delete_joins");

#if !INTERP
            return 1;
#endif
        } // End of Function:  UPI_Entry_Point

#if INTERP
}   /* End of Module hierarchy */
stitch_macro_register();
#endif
```

The following documents are incorporated by cross-reference[

[1] IBM Cu-11 Databook: Macros, March 21 2002.

[2] Universal Serial Bus (USB) Specification Rev1.1, Compaq Computer Corporation, Intel Corporation, Microsoft Corporation, NEC Corporation, September 28 1998.

[3] Synopsys DesignWare USB 1.1 OHCI Host Controller with AHB/PVCI Databook Version 2.6, February 2003.

[4] Open Host Controller Interface Specification (OpenHCI) for USB Rev1.0a, Compaq, Microsoft, National Semiconductor, September 14 1999.

[5] inSilicon TymeWare USB 1.1 Device Controller (UDCVCI) Core User Manual Version 1.1, inSilicon Corporation, November 2000.

[6] Amphion, 2001, *CS6150 Motion JPEG Decoder Databook*, Amphion Semiconductor Ltd.

[7] ANSI/EIA 538-1988, *Facsimile Coding Schemes and Coding Control Functions for Group 4 Facsismile Equipment*, August 1988

[8] Bender, W., P. Chesnais, S. Elo, A. Shaw, and M. Shaw, *Enriching communities: Harbingers of news in the future*, IBM Systems Journal, Vol.35, Nos.3&4, 1996, pp.369-380

[9] CCIR Rec. 601-2, *Encoding Parameters of Digital Television for Studios*, Recommendations of the CCIR, 1990, Vol XI-Part 1, Broadcasting Service (Television), pp.95-104.

[10] Farrell, J., *How to Allocate Bits to Optimize Photographic Image Quality*, Proceedings of IS&T International Conference on Digital Printing Technologies, 1998, pp.572-576

[11] Humphreys, G.W., and V. Bruce, *Visual Cognition*, Lawrence Erlbaum Associates, 1989, p.15

[12] ISO/IEC 19018-1:1994, *Information technology - Digital compression and coding of continuous-tone still images: Requirements and guidelines*, 1994

[13] Lyppens, H., *Reed-Solomon Error Correction*, *Dr. Dobb's Journal* Vol.22, No.1, January 1997

[14] Olsen, J. *Smoothing Enlarged Monochrome Images*, in Glassner, A.S. (ed.), *Graphics Gems*, AP Professional, 1990

[15] Rorabaugh, C, *Error Coding Cookbook*, McGraw-Hill 1996

[16] Thompson, H.S., *Multilingual Corpus 1* CD-ROM, European Corpus Initiative

[17] Urban, S.J., *Review of standards for electronic imaging for facsimile systems*, Journal of Electronic Imaging, Vol.1(1), January 1992, pp.5-21

[18] Wallace, G.K., *The JPEG Still Picture Compression Standard*, Communications of the ACM, Vol.34, No.4, April 1991, pp.30-44

[19] Wicker, S., and Bhargava, V., *Reed-Solomon Codes and their Applications*, IEEE Press 1994

[20] Yasuda, Y., *Overview of Digital Facsimile Coding Techniques in Japan*, Proceedings of the IEEE, Vol. 68(7), July 1980, pp.830-845

[21] SPARC International, *SPARC Architecture Manual, Version8, Revision SAV080SI9308*

[22] Gaisler Research, *The LEON-2 Processor User's Manual, Version 1.0.7*, September 2002

[23] ARM Limited, *AMBA Specification, Rev2.0*, May 1999

[24] ITU-T, Reccomendation T30, Procedures for document facsimile transmission in the general switched telephone network, 07/2003.

[25] Anderson, R, and Kuhn, M., 1997, *Low Cost Attacks on Tamper Resistant Devices*, Security Protocols, Proceedings 1997, LNCS 1361, B. Christianson, B. Crispo, M. Lomas, M. Roe, Eds., Springer-Verlag, pp.125-136.

[26] Anderson, R., and Needham, R.M., *Programming Satan's Computer*, Computer Science Today, LNCS 1000, pp. 426-441.

[27] Atkins, D., Graff, M., Lenstra, A.K., and Leyland, P.C., 1995, *The Magic Words Are Squeamish Ossifrage*, Advances in Cryptology - ASIACRYPT '94 Proceedings, Springer-Verlag, pp. 263-277.

[28] Bains, S., 1997, *Optical schemes tried out in IC test - IBM and Lucent teams take passive and active paths, respectively, to imaging.* EETimes, December 22, 1997.

[29] Bao, F., Deng, R. H., Yan, Y, Jeng, A., Narasimhalu, A.D., Ngair, T., 1997, *Breaking Public Key Cryptosystems on Tamper Resistant Devices in the Presence of Transient Faults*, Security Protocols, Proceedings 1997, LNCS 1361, B. Christianson, B. Crispo, M. Lomas, M. Roe, Eds., Springer-Verlag, pp. 115-124.

[30] Bellare, M., Canetti, R., and Krawczyk. H., 1996, *Keying Hash Functions For Message Authentication*, Advances in Cryptology, Proceedings Crypto'96, LNCS 1109, N. Koblitz, Ed., Springer-Verlag, 1996, pp.1-15. Full version: http://www.research.ibm.com/security/keyed-md5.html

[31] Bellare, M., Canetti, R., and Krawczyk, H., 1996, *The HMAC Construction*, RSA Laboratories CryptoBytes, Vol. 2, No 1, 1996, pp. 12-15.

[32] Bellare, M., Guérin, R., and Rogaway, P., 1995, *XOR MACs: New Methods For Message Authentication Using Finite Pseudorandom Functions*, Advances in Cryptology, Proceedings Crypto'95, LNCS 963, D Coppersmith, Ed., Springer-Verlag, 1995, pp. 15-28.

[33] Blaze, M., Diffie, W., Rivest, R., Schneier, B., Shimomura, T., Thompson, E., Wiener, M., 1996, *Minimal Key Lengths For Symmetric Ciphers To Provide Adequate Commercial Security, A Report By an Ad Hoc Group of Cryptographers and Computer Scientists*, Published on the internet: http://www.livelinks.com/livelinks/bsa/cryptographers.html

[34] Blum, L., Blum, M., and Shub, M., *A Simple Unpredictable Pseudo-random Number Generator*, SIAM Journal of Computing, vol 15, no 2, May 1986, pp 364-383.

[35] Bosselaers, A., and Preneel, B., editors, 1995, *Integrity Primitives for Secure Information Systems: Final Report of RACE Integrity Primitives Evaluation RIPE-RACE 1040*, LNCS 1007, Springer-Verlag, New York.

[36] Brassard, G., 1988, *Modern Cryptography*, a Tutorial, LNCS 325, Springer-Verlag.

[37] Canetti, R., 1997, *Towards Realizing Random Oracles: Hash Functions That Hide All Partial Information, Advances in Cryptology*, Proceedings Crypto'97, LNCS 1294, B. Kaliski, Ed., Springer-Verlag, pp. 455-469.

[38] Cheng, P., and Glenn, R., 1997, *Test Cases for HMAC-MD5 and HMAC-SHA-1*, Network Working Group RFC 2202, http://reference.ncrs.usda.gov/ietf/rfc/2300/rfc2202.htm

[39] Diffie, W., and Hellman, M.E., 1976, *Multiuser Cryptographic Techniques*, AFIPS national Computer Conference, Proceedings '76, pp. 109-112.

[40] Diffie, W., and Hellman, M.E., 1976, *New Directions in Cryptography*, IEEE Transactions on Information Theory, Volume IT-22, No 6 (Nov 1976), pp. 644-654.

[41] Diffie, W., and Hellman, M.E., 1977, *Exhaustive Cryptanalysis of the NBS Data Encryption Standard*, Computer, Volume 10, No 6, (Jun 1977), pp. 74-84.

[42] Dobbertin, H., 1995, *Alf Swindles Ann*, RSA Laboratories CryptoBytes, Volume 1, No 3, p. 5.

[43] Dobbertin, H, 1996, *Cryptanalysis of MD4*, Fast Software Encryption - Cambridge Workshop, LNCS 1039, Springer-Verlag, 1996, pp 53-69.

[44] Dobbertin, H, 1996, *The Status of MD5 After a Recent Attack*, RSA Laboratories CryptoBytes, Volume 2, No 2, pp. 1, 3-6.

[45] Dreifus, H., and Monk, J.T., 1988, *Smart Cards - A Guide to Building and Managing Smart Card Applications*, John Wiley and Sons.

[46] ElGamal, T., 1985, *A Public-Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms*, Advances in Cryptography, Proceedings Crypto'84, LNCS 196, Springer-Verlag, pp. 10-18.

[47] ElGamal, T., 1985, *A Public-Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms*, IEEE Transactions on Information Theory, Volume 31, No 4, pp. 469-472

[48] Feige, U., Fiat, A, and Shamir, A., 1988, *Zero Knowledge Proofs of Identity*, J Cryptography, Volume 1, pp. 77-904.

[49] Feigenbaum, J., 1992, *Overview of Interactive Proof Systems and Zero-Knowledge*, Contemporary Cryptology - The Science of Information Integrity, G Simmons, Ed., IEEE Press, New York.

[50] FIPS 46-1, 1977, *Data Encryption Standard*, NIST, US Department of Commerce, Washington D.C., Jan 1977.

[51] FIPS 180, 1993, *Secure Hash Standard*, NIST, US Department of Commerce, Washington D.C., May 1993.

[52] FIPS 180-1, 1995, *Secure Hash Standard*, NIST, US Department of Commerce, Washington D.C., April 1995.

[53] FIPS 186, 1994, *Digital Signature Standard*, NIST, US Department of Commerce, Washington D.C., 1994.

[54] Gardner, M., 1977, *A New Kind of Cipher That Would Take Millions of Years to Break*, Scientific American, Vol. 237, No. 8, pp. 120-124.

[55] Girard, P., Roche, F. M., Pistoulet, B., 1986, *Electron Beam Effects on VLSI MOS: Conditions for Testing and Reconfiguration*, Wafer-Scale Integration, G. Saucier and J. Trihle, Eds., Amsterdam.

[56] Girard, P., Pistoulet, B., Valenza, M., and Lorival, R., 1987, *Electron Beam Switching of Floating Gate MOS Transistors*, IFIP International Workshop onWafer Scale International, Brunel University, Sept. 23-25, 1987.

[57] Goldberg, I., and Wagner, D., 1996, *Randomness and the Netscape Browser*, Dr. Dobb's Journal, January 1996.

[58] Guilou, L. G., Ugon, M., and Quisquater, J., 1992, *The Smart Card, Contemporary Cryptology - The Science of Information Integrity*, G Simmons, Ed., IEEE Press, New York.

[59] Gutman, P., 1996, *Secure Deletion of Data From Magnetic and Solid-State Memory*, Sixth USENIX Security Symposium Proceedings (July 1996), pp. 77-89.

[60] Hendry, M., 1997, *Smart Card Security and Applications*, Artech House, Norwood MA.

[61] Holgate, S. A., 1998, *Sensing is Believing*, New Scientist, 15 August 1998, p 20.

[62] Johansson, T., 1997, *Bucket Hashing with a Small Key Size*, Advances in Cryptology, Proceedings Eurocrypt'97, LNCS 1233, W. Fumy, Ed., Springer-Verlag, pp. 149-162.

[63] Kahn, D., 1967, *The Codebreakers: The Story of Secret Writing*, New York: Macmillan Publishing Co.

[64] Kaliski, B., 1991, *Letter to NIST regarding DSS*, 4 Nov 1991.

[65] Kaliski, B., 1998, *New Threat Discovered and Fixed*, RSA Laboratories Web site http://www.rsa.com/rsalabs/pkcs1

[66] Kaliski, B., and Robshaw, M, 1995, *Message Authentication With MD5*, RSA Laboratories Crypto-Bytes, Volume 1, No 1, pp. 5-8.

[67] Kaliski, B., and Yin, Y.L., 1995, *On Differential and Linear Cryptanalysis of the RC5 Encryption Algorithm*, Advances in Cryptology, Proceedings Crypto '95, LNCS 963, D. Coppersmith, Ed., Springer-Verlag, pp. 171-184.

[68] Klapper, A., and Goresky, M., 1994, *2-Adic Shift Registers*, Fast Software Encryption: Proceedings Cambridge Security Workshop '93, LNCS 809, R. Anderson, Ed., Springer-Verlag, pp. 174-178.

[69] Klapper, A., 1996, *On the Existence of Secure Feedback Registers*, Advances in Cryptology, Proceedings Eurocrypt'96, LNCS 1070, U. Maurer, Ed., Springer-Verlag, pp. 256-267.

[70] Kleiner, K., 1998, *Cashing in on the not so smart cards*, New Scientist, 20 June 1998, p 12.

[71] Knudsen, L.R., and Lai, X., *Improved Differential Attacks on RC5*, Advances in Cryptology, Proceedings Crypto'96, LNCS 1109, N. Koblitz, Ed., Springer-Verlag, 1996, pp.216-228

[72] Knuth, D.E., 1998, *The Art of Computer Programing - Volume 2/ Seminumerical Algorithms*, 3rd edition, Addison-Wesley.

[73] Krawczyk, H., 1995, *New Hash Functions for Message Authentication*, Advances in Cryptology, Proceedings Eurocrypt'95, LNCS 921, L Guillou, J Quisquater, (editors), Springer-Verlag, pp. 301-310.

[74] Krawczyk, H., 199x, *Network Encryption - History and Patents*, internet publication: http://www.cygnus.com/~gnu/netcrypt.html

[75] Krawczyk, H., Bellare, M, Canetti, R., 1997, *HMAC: Keyed Hashing for message Authentication*, Network Working Group RFC 2104, http://reference.ncrs.usda.gov/ietf/rfc/2200/rfc2104.htm

[76] Lai, X., 1992, *On the Design and Security of Block Ciphers*, ETH Series in Information Processing, J.L. Massey (editor), Volume 1, Konstanz: hartung-Gorre Verlag (Zurich).

[77] Lai, X, and Massey, 1991, J.L, *A Proposal for a New Block Encryption Standard*, Advances in Cryptology, Proceedings Eurocrypt'90, LNCS 473, Springer-Verlag, pp. 389-404.

[78] Massey, J.L., 1969, *Shift Register Sequences and BCH Decoding*, IEEE Transactions on Information Theory, IT-15, pp. 122-127.

[79] Mende, B., Noll, L., and Sisodiya, S., 1997, *How Lavarand Works*, Silicon Graphics Incorporated, published on Internet: http://lavarand.sgi.com (also reported in Scientific American, November 1997 p. 18, and New Scientist, 8 November 1997).

[80] Menezes, A. J., van Oorschot, P. C., Vanstone, S. A., 1997, *Handbook of Applied Cryptography*, CRC Press.

[81] Merkle, R.C., 1978, *Secure Communication Over Insecure Channels*, Communications of the ACM, Volume 21, No 4, pp. 294-299.

[82] Montgomery, P. L., 1985, *Modular Multiplication Without Trial Division*, Mathematics of Computation, Volume 44, Number 170, pp. 519-521.

[83] Moreau, T., *A Practical "Perfect" Pseudo-Random Number Generator*, paper submitted to Computers in Physics on February 27 1996, Internet version: http://www.connotech.com/BBS.HTM

[84] Moreau, T., 1997, *Pseudo-Random Generators, a High-Level Survey-in-Progress*, Published on the internet: http://www.cabano.com/connotech/RNG.HTM

[85] NIST, 1994 , *Digital Signature Standard*, NIST ISL Bulletin, online version at http://csrc.ncsl.nist.gov/nistbul/cs194-11.txt

[86] Oehler, M., Glenn, R., 1997, *HMAC-MD5 IP Authentication with Replay Prevention*, Network Working Group RFC 2085, http://reference.ncrs.usda.gov/ietf/rfc/2100/rfc2085.txt

[87] Oppliger, R., 1996, *Authentication Systems For Secure Networks*, Artech House, Norwood MA.

[88] Preneel, B., van Oorschot, P.C., 1996, *MDx-MAC And Building Fast MACs From Hash Functions*, Advances in Cryptology, Proceedings Crypto'95, LNCS 963, D. Coppersmith, Ed., Springer-Verlag, pp. 1-14.

[89] Preneel, B., van Oorschot, P.C., 1996, *On the Security of Two MAC Algorithms*, Advances in Cryptology, Proceedings Eurocrypt'96, LNCS 1070, U. Maurer, Ed., Springer-Verlag, 1996, pp. 19-32.

[90] Preneel, B., Bosselaers, A., Dobbertin, H., 1997, *The Cryptographic Hash Function RIPEMD-160*, CryptoBytes, Volume 3, No 2, 1997, pp. 9-14.

[91] Rankl, W., and Effing, W., 1997, *Smart Card Handbook*, John Wiley and Sons (first published as Handbuch der Chipkarten, Carl Hanser Verlag, Munich, 1995).

[92] Ritter, T., 1991, *The Efficient Generation of Cryptographic Confusion Sequences*, Cryptologia, Volume 15, No 2, pp. 81-139.

[93] Rivest, R.L, 1993, *Dr. Ron Rivest on the Difficulties of Factoring*, Ciphertext: The RSA Newsletter, Vol 1, No 1, pp. 6, 8.

[94] Rivest, R.L., 1991, *The MD4 Message-Digest Algorithm*, Advances in Cryptology, Proceedings Crypto'90, LNCS 537, S. Vanstone, Ed., Springer-Verlag, pp. 301-311.

[95] Rivest, R.L., 1992, *The RC4 Encryption Algorithm*, RSA Data Security Inc. (This document has not been made public).

[96] Rivest, R.L., 1992, *The MD4 Message-Digest Algorithm*, Request for Comments (RFC) 1320, Internet Activities Board, Internet Privacy Task Force, April 1992.

[97] Rivest, R.L., 1992, *The MD5 Message-Digest Algorithm*, Request for Comments (RFC) 1321, Internet Activities Board, Internet privacy Task Force.

[98] Rivest, R.L., 1995, *The RC5 Encryption Algorithm*, Fast Software Encryption, LNCS 1008, Springer-Verlag, pp. 86-96.

[99] Rivest, R.L., Shamir, A., and Adleman, L.M., 1978, A *Method For Obtaining Digital Signatures and Public-Key Cryptosystems*, Communications of the ACM, Volume 21, No 2, pp. 120-126.

[100] Schneier, S., 1994, *Description of a New Variable-Length Key, 64-Bit Block Cipher (Blowfish)*, Fast Software Encryption (December 1993), LNCS 809, Springer-Verlag, pp. 191-204.

[101] Schneier, S., 1995, *The Blowfish Encryption Algorithm - One Year Later*, Dr Dobb's Journal, September 1995.

[102] Schneier, S., 1996, *Applied Cryptography*, Wiley Press.

[103] Schneier, S., 1998, *The Blowfish Encryption Algorithm*, revision date February 25, 1998, http://www.counterpane.com/blowfish.html

[104] Schneier, S., 1998, *The Crypto Bomb is Ticking*, Byte Magazine, May 1998, pp. 97-102.

[105] Schnorr, C.P., 1990, *Efficient Identification and Signatures for Smart Cards*, Advances in Cryptology, Proceedings Eurocrypt'89, LNCS 435, Springer-Verlag, pp. 239-252.

[106] Shamir, A., and Fiat, A., Method, *Apparatus and Article For Identification and Signature*, U.S. Patent number 4,748,668, 31 May 1988.

[107] Shor, W., 1994, *Algorithms for Quantum Computation: Discrete Logarithms and Factoring*, Proc. 35th Symposium. Foundations of Computer Science (FOCS), IEEE Computer Society, Los Alamitos, Calif., 1994.

[108] Simmons, G. J., 1992, A *Survey of Information Authentication*, Contemporary Cryptology - The Science of Information Integrity, G Simmons, Ed., IEEE Press, New York.

[109] Tewksbury, S. K., 1998, *Architectural Fault Tolerance, Integrated Circuit Manufacturability*, Pineda de Gyvez, J., and Pradhan, D. K., Eds., IEEE Press, New York.

[110] TSMC, 2000, *SFC0008_08B9_HE*, 8K × 8 Embedded Flash Memory Specification, Rev 0.1.

[111] Tsudik, G., 1992, *Message Authentication With One-way Hash Functions*, Proceedings of Infocom '92 (Also in Access Control and Policy Enforcement in Internetworks, Ph.D. Dissertation, Computer Science Department, University of Southern California, April 1991).

[112] Vallett. D., Kash, J., and Tsang, J., *Watching Chips Work*, IBM MicroNews, Vol 4, No 1, 1998.

[113] Vazirani, U.V., and Vazirani, V.V., 1984, *Efficient and Secure Random Number Generation*, 25th Symposium. Foundations of Computer Science (FOCS), IEEE Computer Society, 1984, pp. 458-463.

[114] Wagner, D., Goldberg, I., and Briceno, M., 1998, *GSM Cloning*, ISAAC Research Group, University of California, http://www.isaac.cs.berkeley.edu/isaac/gsm-faq.html

[115] Wiener, M.J., 1997, *Efficient DES Key Search - An Update*, RSA Laboratories CryptoBytes, Volume 3, No 2, pp. 6-8.

[116] Zoreda, J.L., and Otón, J.M., 1994, *Smart Cards*, Artech House, Norwood MA.

[117] Authentication Protocols v0_2 29 November 2002. Silverbrook Research.

[118] H. Krawczyk IBM, M. Bellare UCSD, R. Canetti IBM, RFC 2104, February 1997, http://www.ietf.org/rfc/rfc2104.txt

[119] 4-3-1-2-QAChipSpec v4_09 17 Apr. 2003,Silverbrook Research

[120] 4-4-9-4 SoPEC_hardware_design_v3_1 28 Jan. 2003

[121] Silverbrook Research, 2002, *4-3-1-8 QID Requirements Specification*.

[122]   TSMC, Oct 1, 2000, *SFC0008_08B9_HE*, 8K × 8 Embedded Flash Memory Specification, Rev 0.1.

[123]   TSMC (design service division), Sep 10, 2001, *0.25um Embedded Flash Test Mode User Guide*, V0.3.

[124]   TSMC (EmbFlash product marketing), Oct 19, 2001, *0.25um Application Note*, V2.2.

[125]   Artisan Components, Jan 99, Process Perfect Library Databook 2.5-Volt Standard Cells, Rev1.0.

[126]   "4-3-1-2 QA Chip Technical Reference v 4.06", Silverbrook Research.

[127]   National Institute of Standards and Technology - Federal Information Processing Standardshttp://csrc.nist.gov/publications/fips/

[128]   United States Patent Application No. 09/575,108, filed May 23, 2000, Silverbrook Research Pty Ltd

[129]   United States Patent Application No. 09/575,109, filed May 23, 2000, Silverbrook Research Pty Ltd

[130]   United States Patent Application No. 09/575,100, filed May 23, 2000, Silverbrook Research Pty Ltd

[131]   United States Patent Application No. 09/607,985, filed May 23, 2000, Silverbrook Research Pty Ltd

[132]   United States Patent No. 6,398,332, filed June 30, 2000, Silverbrook Research Pty

[133]   United States Patent No. 6,394,573, filed June 30, 2000, Silverbrook Research Pty Ltd

[134]   United States Patent No. 6,622,923, filed June 30, 2000, Silverbrook Research Pty Ltd

[135]   United States Patent Application No. 09/517,539, filed March 2, 2000, Silverbrook Research Pty Ltd

[136]   United States Patent No.6,566,858, filed July 10, 1998, Silverbrook Research Pty Ltd

[137]   United States Patent No.6,331,946, filed July 10, 1998, Silverbrook Research Pty Ltd

[138]   United States Patent No 6,246,970, filed July 10, 1998, Silverbrook Research Pty Ltd

[139]   United States Patent No.6,442,525, filed July 10, 1998, Silverbrook Research Pty Ltd

[140]   United States Patent Application No.09/517,384, filed March 2, 2000, Silverbrook Research Pty Ltd

[141]   United States Patent Application No.09/505,951, filed February 21, 2001, Silverbrook Research Pty Ltd

[142]   United States Patent No 6,374,354, filed March 2, 2000, Silverbrook Research Pty Ltd

[143]   United States Patent Application No.09/517,608, filed March 2, 2000, Silverbrook Research Pty Ltd

[144]   United States Patent No 6,334,190, filed March 2, 2000, Silverbrook Research Pty Ltd

[145]   United States Patent Application No.09/517,541, filed March 2, 2000, Silverbrook Research Pty Ltd